SnakeMD

The Renegade Coder

Nov 22, 2023

CONTENTS

SnakeMD is a library for generating markdown files using Python. Use the links below to navigate the docs.

CHAPTER

ONE

INSTALLATION

SnakeMD is fairly hassle-free and can be installed like most third-party libraries using pip. Only caveat worth noting is which SnakeMD version is compatibile with your personal version of Python. See below for more details.

1.1 Python Support

SnakeMD at its core is a dependency-free markdown generation library. As a result, you shouldn't have to concern yourself with issues that can arise due to dependency conflicts in your environments. However, SnakeMD is a Python library and is constrained by the various versions of Python. To help you navigate this, the SnakeMD documentation includes a table of Python support as seen below. Make sure to always install a version of SnakeMD that is tested for your version of Python.

Python	3.11	3.10	3.9	3.8
SnakeMD >= 2.0	Yes	Yes	Yes	Yes
SnakeMD 0.12 - 0.15	Yes	Yes	Yes	Yes
SnakeMD < 0.12		Yes	Yes	Yes

1.2 Basic Installation

The quick and dirty way to install SnakeMD is to use pip:

```
pip install snakemd
```

If you'd like access to any pre-releases, you can also install SnakeMD with the `--pre` flag:

```
pip install --pre snakemd
```

Be aware that pre-releases are not suitable for production code.

1.3 Building From Source

For folks who want to help with development, we generally recommend the following workflow as of v2.1.0 (see previous version of docs for older guides):

1.3.1 1. Clone the Sourcecode From GitHub

To start, we can download the sourcecode by running a git clone command.

```
PS E:\Projects> git clone https://github.com/TheRenegadeCoder/SnakeMD.git
Cloning into 'SnakeMD'...
remote: Enumerating objects: 1477, done.
remote: Counting objects: 100% (63/63), done.
remote: Compressing objects: 100% (50/50), done.
remote: Total 1477 (delta 27), reused 27 (delta 12), pack-reused 1414
Receiving objects: 100% (1477/1477), 6.43 MiB | 5.68 MiB/s, done.
Resolving deltas: 100% (814/814), done.git clone https://github.com/TheRenegadeCoder/
↪SnakeMD.git
```

1.3.2 2. Change Directories

With the sourcecode downloaded, we can now navigate to the project folder.

```
PS E:\Projects> cd SnakeMD

PS E:\Projects\SnakeMD>
```

1.3.3 3. Initialize the Repo With Poetry

Assuming you have poetry installed, you can immediately get up to speed by running the install command.

```
PS E:\Projects> poetry install
```

1.3.4 4. Verify Everything Works

A quick way to check if everything worked out is to try to run the tests.

```
PS E:\Projects\SnakeMD> poetry run pytest
============================= test session starts ==============================
platform win32 -- Python 3.11.3, pytest-7.3.1, pluggy-1.0.0
rootdir: E:\Projects\SnakeMD
configfile: pyproject.toml
testpaths: tests
collected 168 items

tests\test_code.py .....                                                 [  2%]
tests\test_document.py ........................                          [ 17%]
tests\test_heading.py .................                                  [ 27%]
```

(continues on next page)

(continued from previous page)

```
tests\test_horizontal_rule.py .                                          [ 27%]
tests\test_inline.py ....................................                [ 52%]
tests\test_md_list.py ......................                             [ 67%]
tests\test_module.py .                                                   [ 68%]
tests\test_paragraph.py ................                                 [ 79%]
tests\test_quote.py ........                                             [ 84%]
tests\test_raw.py ....                                                   [ 86%]
tests\test_table.py ..............                                       [ 95%]
tests\test_table_of_contents.py .......                                  [100%]

============================== 168 passed in 0.15s ==============================
```

And at the same time, why not verify that docs can be constructed:

```
PS E:\Projects\SnakeMD> poetry run sphinx-build -b dirhtml docs docs/_build
Running Sphinx v6.2.1
loading intersphinx inventory from https://docs.python.org/3/objects.inv...
building [mo]: targets for 0 po files that are out of date
writing output...
building [dirhtml]: targets for 9 source files that are out of date
updating environment: [new config] 9 added, 0 changed, 0 removed
reading sources... [100%] version-history
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] version-history
generating indices... genindex py-modindex done
writing additional pages... search done
copying static files... done
copying extra files... done
dumping search index in English (code: en)... done
dumping object inventory... done
build succeeded.

The HTML pages are in docs\_build.
```

If you see anything like above, you're ready to start development.

CHAPTER

TWO

USAGE

SnakeMD is a Python library for building markdown documents. We can use it by importing the SnakeMD module into our program directly:

```
>>> import snakemd
```

This way, we'll have access to all of the classes available in the SnakeMD module. From here, we can take advantage of a handy function to create a new document:

```
>>> doc = snakemd.new_doc()
```

This will create a new *snakemd.Document* object. Alternatively, we can import the Document class directly:

```
>>> from snakemd import Document
```

From here, we can instantiate the Document class:

```
>>> doc = Document()
```

While there is nothing in our document currently, we can render an empty one as follows:

```
>>> doc.dump("README")
```

This will create an empty README.md file in our working directory. Of course, if we want something more interesting, we'll have to add some content to our document. To start, we'll add a title to the document:

```
>>> doc.add_heading("Why Use SnakeMD?")
Heading(text=[Inline(text='Why Use SnakeMD?',...)], level=1)
```

From here, we can do pretty much anything we'd like. Some quick actions might be to include a paragraph about this library as well as a list of reasons why we might use it:

```
>>> p = doc.add_paragraph(
...     """
...     SnakeMD is a library for generating markdown, and here's
...     why you might choose to use it:
...     """
... )

>>> doc.add_unordered_list([
...     "SnakeMD makes it easy to create markdown files.",
...     "SnakeMD has been used to automate documentation for The Renegade Coder projects.
↪"
```

(continues on next page)

(continued from previous page)

```
... ])
MDList(items=[...], ordered=False, checked=None)
```

One thing that's really cool about using SnakeMD is that we can build out the structure of a document before we modify it to include any links. For example, notice how we saved the output of the *snakemd.Document.add_paragraph()* method from above. Well, as it turns out, all of the document methods return the objects that are generated as a result of their use. In this case, the method returns a Paragraph object which we can modify. Here's how we might insert a link to the docs:

```
>>> p.insert_link("SnakeMD", "https://snakemd.therenegadecoder.com")
Paragraph(content=[...])
```

And if all goes well, we can output the results by outputting the document like before. Or, if we just need to see the results as a string, we can convert the document to a string directly:

```
>>> print(doc)
```

And this is what we'll get:

```
# Why Use SnakeMD?

[SnakeMD](https://snakemd.therenegadecoder.com) is a library for generating markdown,␣
↪and here's why you might choose to use it:

- SnakeMD makes it easy to create markdown files.
- SnakeMD has been used to automate documentation for The Renegade Coder projects.
```

For completion, here is a working program to generate the document from above in a file called README.md:

```
import snakemd

doc = snakemd.new_doc()

doc.add_heading("Why Use SnakeMD?")
p = doc.add_paragraph(
  """
  SnakeMD is a library for generating markdown, and here's
  why you might choose to use it:
  """
)
doc.add_unordered_list([
    "SnakeMD makes it easy to create markdown files.",
    "SnakeMD has been used to automate documentation for The Renegade Coder projects."
])
p.insert_link("SnakeMD", "https://snakemd.therenegadecoder.com")

doc.dump("README")
```

As always, feel free to check out the rest of the documentation for all of the ways you can make use of SnakeMD. If you find an issues, make sure to head over to the GitHub repo and let us know.

CHAPTER

THREE

DOCUMENTATION

The documentation page lists out all of the relevant classes and functions for generating markdown documents in Python.

3.1 The Document API

SnakeMD is designed with different types of users in mind. The main type of user is the person who wants to design and generate markdown quickly without worrying too much about the format or styling of their documents. To help this type of user, we've developed a high-level API which consists of a single function, *snakemd.new_doc()*. This function returns a *snakemd.Document* object that is ready to be modified using any of the convenience methods available in the *snakemd.Document* class. Both the *snakemd.new_doc()* function and the *snakemd.Document* class are detailed below.

3.1.1 Module

The SnakeMD module contains all of the functionality for generating markdown files with Python. To get started, check out *Usage* for quickstart sample code.

The SnakeMD module is the root module of the snakemd system. It imports all classes to be used directly through snakemd, so users don't need to know the underlying directory structure. Likewise, directory structure can be changed in future iterations of the project without affecting users.

snakemd.**new_doc**() → *Document*

Creates a new SnakeMD document. This is a convenience function that allows you to create a new markdown document without having to import the *Document* class. This is useful for anyone who wants to take advantage of the procedural interface of SnakeMD. For those looking for a bit more control, each element class will need to be imported as needed.

```
>>> doc = snakemd.new_doc()
```

Returns
a new Document object

3.1.2 Document

Note: All of the methods described in the *snakemd.Document* class are assumed to work without any *snakemd.Element* imports. In circumstances where methods may make use of Elements, such as in *snakemd.Document.add_table()*, the snakemd module will be referenced directly in the sample source code.

For the average user, the document object is the only object in the library necessary to create markdown files. Document objects are automatically created from the *new_doc()* function of the SnakeMD module.

class snakemd.**Document**(*elements:* *list*[snakemd.elements.Element] = *None*)

Bases: object

A document represents a markdown file. Documents store a collection of elements which are appended with new lines between to generate the markdown document. Document methods are intended to provided convenience when generating a markdown file. However, the functionality is not exhaustive. To get the full range of markdown functionality, you can take advantage of the *add_block()* function to provide custom markdown blocks.

Parameters

elements (*list*[Element]) – an optional list of elements that make up a markdown document

New in version 2.2: Included to make __repr__ more useful

__repr__() → str

Renders self as an unambiguous string for development. In this case, it displays in the style of a dataclass, where instance variables are listed with their values.

```
>>> doc = snakemd.new_doc()
>>> repr(doc)
'Document(elements=[])'
```

Returns

the MDList object as a development string

__str__() → str

Renders the markdown document from a list of elements.

```
>>> doc = snakemd.new_doc()
>>> doc.add_heading("First")
Heading(text=[...], level=1)
>>> print(doc)
# First
```

Returns

the document as a markdown string

add_block(*block:* Block) → *Block*

A generic function for appending blocks to the document. Use this function when you want a little more control over what the output looks like.

```
>>> doc = snakemd.new_doc()
>>> doc.add_block(snakemd.Heading("Python is Cool!", 2))
Heading(text=[Inline(text='Python is Cool!',...)], level=2)
```

(continues on next page)

(continued from previous page)

```
>>> print(doc)
## Python is Cool!
```

Parameters
block (Block) – a markdown block (e.g., Table, Heading, etc.)

Returns
the *Block* added to this Document

add_checklist(*items: Iterable[str]*) → *MDList*

A convenience method which adds a checklist to the document.

```
>>> doc = snakemd.new_doc()
>>> doc.add_checklist(["Okabe", "Mayuri", "Kurisu"])
MDList(items=[...], ordered=False, checked=False)
>>> print(doc)
- [ ] Okabe
- [ ] Mayuri
- [ ] Kurisu
```

Parameters
items (*Iterable[str]*) – a "list" of strings

Returns
the *MDList* added to this Document

add_code(*code: str*, *lang: str = 'generic'*) → *Code*

A convenience method which adds a code block to the document:

```
>>> doc = snakemd.new_doc()
>>> doc.add_code("x = 5")
Code(code='x = 5', lang='generic')
>>> print(doc)
```generic
x = 5
```
```

Parameters

- **code** (*str*) – a preformatted code string
- **lang** (*str*) – the language for syntax highlighting

Returns
the *Code* block added to this Document

add_heading(*text: str*, *level: int = 1*) → *Heading*

A convenience method which adds a heading to the document:

```
>>> doc = snakemd.new_doc()
>>> doc.add_heading("Welcome to SnakeMD!")
Heading(text=[Inline(text='Welcome to SnakeMD!',...)], level=1)
```

(continues on next page)

(continued from previous page)

```
>>> print(doc)
# Welcome to SnakeMD!
```

Parameters

- **text** (*str*) – the text for the heading
- **level** (*int*) – the level of the heading from 1 to 6

Returns

the *Heading* added to this Document

add_horizontal_rule() → *HorizontalRule*

A convenience method which adds a horizontal rule to the document:

```
>>> doc = snakemd.new_doc()
>>> doc.add_horizontal_rule()
HorizontalRule()
>>> print(doc)
***
```

Returns

the *HorizontalRule* added to this Document

add_ordered_list(*items: Iterable[str]*) → *MDList*

A convenience method which adds an ordered list to the document:

```
>>> doc = snakemd.new_doc()
>>> doc.add_ordered_list(["Goku", "Piccolo", "Vegeta"])
MDList(items=[...], ordered=True, checked=None)
>>> print(doc)
1. Goku
2. Piccolo
3. Vegeta
```

Parameters

items (*Iterable[str]*) – a "list" of strings

Returns

the *MDList* added to this Document

add_paragraph(*text: str*) → *Paragraph*

A convenience method which adds a paragraph of text to the document:

```
>>> doc = snakemd.new_doc()
>>> doc.add_paragraph("Mitochondria is the powerhouse of the cell.")
Paragraph(content=[...])
>>> print(doc)
Mitochondria is the powerhouse of the cell.
```

Parameters

text (*str*) – any arbitrary text

Returns
the *Paragraph* added to this Document

add_quote(*text: str*) → *Quote*

A convenience method which adds a blockquote to the document:

```
>>> doc = snakemd.new_doc()
>>> doc.add_quote("Welcome to the Internet!")
Quote(content=[Raw(text='Welcome to the Internet!')])
>>> print(doc)
> Welcome to the Internet!
```

Parameters
text (*str*) – the text to be quoted

Returns
the *Quote* added to this Document

add_raw(*text: str*) → *Raw*

A convenience method which adds text as-is to the document:

```
>>> doc = snakemd.new_doc()
>>> doc.add_raw("X: 5\nY: 4\nZ: 3")
Raw(text='X: 5\nY: 4\nZ: 3')
>>> print(doc)
X: 5
Y: 4
Z: 3
```

Parameters
text (*str*) – some text

Returns
the *Raw* block added to this Document

add_table(*header: Iterable[str]*, *data: Iterable[Iterable[str]]*, *align: Iterable[*Align*] = None*, *indent: int = 0*) → *Table*

A convenience method which adds a table to the document:

```
>>> doc = snakemd.new_doc()
>>> header = ["Place", "Name"]
>>> rows = [["1st", "Robert"], ["2nd", "Rae"]]
>>> align = [snakemd.Table.Align.CENTER, snakemd.Table.Align.RIGHT]
>>> doc.add_table(header, rows, align=align)
Table(header=[...], body=[...], align=[...], indent=0)
>>> print(doc)
| Place | Name   |
| :---: | -----: |
| 1st   | Robert |
| 2nd   | Rae    |
```

Parameters

- **header** (*Iterable[str]*) – a “list” of strings

- **data** (*Iterable[Iterable[str]]*) – a "list" of "lists" of strings
- **align** (*Iterable[Table.Align]*) – a "list" of column alignment values; defaults to None
- **indent** (*int*) – indent size for the whole table

Returns
the *Table* added to this Document

add_table_of_contents(*levels: range = range(2, 3)*) → *TableOfContents*

A convenience method which creates a table of contents. This function can be called where you want to add a table of contents to your document. The table itself is lazy loaded, so it always captures all of the heading blocks regardless of where the table of contents is added to the document.

```
>>> doc = snakemd.new_doc()
>>> doc.add_table_of_contents()
TableOfContents(levels=range(2, 3))
>>> doc.add_heading("First Item", 2)
Heading(text=[Inline(text='First Item',...)], level=2)
>>> doc.add_heading("Second Item", 2)
Heading(text=[Inline(text='Second Item',...)], level=2)
>>> print(doc)
1. [First Item](#first-item)
2. [Second Item](#second-item)

## First Item

## Second Item
```

Parameters
levels (*range*) – a range of heading levels to be included in the table of contents

Returns
the *TableOfContents* added to this Document

add_unordered_list(*items: Iterable[str]*) → *MDList*

A convenience method which adds an unordered list to the document.

```
>>> doc = snakemd.new_doc()
>>> doc.add_unordered_list(["Deku", "Bakugo", "Kirishima"])
MDList(items=[...], ordered=False, checked=None)
>>> print(doc)
- Deku
- Bakugo
- Kirishima
```

Parameters
items (*Iterable[str]*) – a "list" of strings

Returns
the *MDList* added to this Document

dump(*name: str*, *directory: str | PathLike = ''*, *ext: str = 'md'*, *encoding: str = 'utf-8'*) → None

Outputs the markdown document to a file. This method assumes the output directory is the current working directory. Any alternative directory provided will be made if it does not already exist. This method also assumes a file extension of md and a file encoding of utf-8, all of which are configurable through the method parameters.

```
>>> doc = snakemd.new_doc()
>>> doc.add_horizontal_rule()
HorizontalRule()
>>> doc.dump("README")
```

Parameters

- **name** (*str*) – the name of the markdown file to output without the file extension
- **directory** (*str | os.PathLike*) – the output directory for the markdown file; defaults to “”

  Changed in version 2.2: Renamed from dir to directory to avoid built-in clashes
- **ext** (*str*) – the output file extension; defaults to “md”
- **encoding** (*str*) – the encoding to use; defaults to utf-8

get_elements() → list[*snakemd.elements.Element*]

A getter method which allows the user to retrieve the underlying document structure of elements as a list. The return value is directly aliased to the underlying representation, so any changes to this object will change the document.

The primary use of this method is to share an alias to the underlying document structure to other useful components like TableOfContents without creating circular references.

New in version 2.2: Included as a part of the TableOfContents rework

Returns

the list of block comprising this document

scramble() → None

A silly method which mixes all of the blocks in this document in a random order.

```
>>> doc = snakemd.new_doc()
>>> doc.add_horizontal_rule()
HorizontalRule()
>>> doc.scramble()
>>> print(doc)
***
```

3.2 The Element API

For users who want a little more control over how their markdown is formatted, SnakeMD provides a low-level API constructed of elements.

3.2.1 Element Interface

Broadly speaking, anything that can be rendered as markdown is known as an element. Below is the element interface.

class snakemd.**Element**

Bases: ABC

A generic element interface which provides a framework for all types of elements in the collection. In short, elements must be able to be converted to their markdown representation using the built-in `str` constructor. They must also be able to be converted into development strings using the `repr()` function.

abstract __repr__() → str

The developer's string method to help make sense of objects. For the purposes of this repo, the __repr__ method should create strings that can be used to recreate the element, much like the built-in feature of dataclasses (a feature which may be adopted in future versions of snakemd). Ultimately, this method must be implemented by all inheriting classes.

Returns
an unambiguous representation of the element

abstract __str__() → str

The default string method to be implemented by all inheriting classes.

Returns
a markdown ready representation of the element

For consistency, element mutators all return self to allow for method chaining. This is sometimes referred to as the fluent interface pattern, and it's particularly useful for applying a series of changes to a single element. This design choice most obviously shines in both *snakemd.Paragraph*, which allows different aspects of the text to be replaced over a series of chained methods, and *snakemd.Inline*, which allows inline elements to be styled over a series of chained methods.

For practical purposes, elements cannot be constructed directly. Instead, they are broken down into two main categories: block and inline.

3.2.2 Block Elements

SnakeMD block elements borrow from the idea of block-level elements from HTML. And because Markdown documents are constructed from a series of blocks, users of SnakeMD can seemlessly append their own custom blocks using the *snakemd.Document.add_block()* method. To make use of this method, blocks must be imported and constructed manually, like the following *snakemd.Heading* example:

```
>>> from snakemd import Heading, new_doc
>>> doc = new_doc()
>>> heading = doc.add_block(Heading("Hello, World!", 2))
```

The remainder of this section introduces the Block interface as well as all of the Blocks currently available for use.

Block Interface

All markdown blocks inherit from the Block interface.

class snakemd.**Block**

Bases: *Element*

A block element in Markdown. A block is defined as a standalone element starting on a newline. Examples of blocks include paragraphs (i.e., <p>), headings (e.g., <h1>, <h2>, etc.), tables (i.e., <table>), and lists (e.g., <ol>, <ul>, etc.).

Code

class snakemd.**Code**(*code: str* | Code, *lang: str* = *'generic'*)

Bases: *Block*

A code block is a standalone block of syntax-highlighted code. Code blocks can have generic highlighting or highlighting based on their language.

Parameters

- **code** (*str* | Code) – the sourcecode to format as a Markdown code block
  - set to a string to render a preformatted code block (i.e., whitespace is respected)
  - set to a Code object to render a nested code block
- **lang** (*str*) – the programming language for the code block; defaults to 'generic'

__repr__() → str

Renders self as an unambiguous string for development. In this case, it displays in the style of a dataclass, where instance variables are listed with their values.

```
>>> code = Code('x = 87')
>>> repr(code)
"Code(code='x = 87', lang='generic')"
```

Returns
the Code object as a development string

__str__() → str

Renders the code block as a markdown string. Markdown code blocks are returned with the fenced code block format using backticks:

```
```python
x = 5
y = 2 + x
```
```

Code blocks can be nested and will be rendered with increasing numbers of backticks.

Returns
the code block as a markdown string

Heading

class snakemd.**Heading**(*text: str | Inline | Iterable[Inline | str]*, *level: int*)

Bases: *Block*

A heading is a text block which serves as the title for a new section of a document. Headings come in six main sizes which correspond to the six headings sizes in HTML (e.g., <h1>).

Raises
ValueError – when level < 1 or level > 6

Parameters

- **text** (*str | Inline | Iterable[Inline | str]*) – the heading text
  - set to a string to render raw heading text
  - set to an Inline object to render a styled heading (e.g., bold, link, code, etc.)
  - set to a "list" of the prior options to render a header with more granular control over the individual inline elements
- **level** (*int*) – the heading level between 1 and 6

__repr__() → str

Renders self as an unambiguous string for development. In this case, it displays in the style of a dataclass, where instance variables are listed with their values.

Note that Headings can accept a variety of string-like inputs. However, the underlying representation forces all possible inputs to be a list of Inline objects. As a result, the repr representation will often be significantly more complex than expected.

```
>>> heading = Heading("", 1)
>>> repr(heading)
"Heading(text=[Inline(text='',...)], level=1)"
```

Returns
the Code object as a development string

__str__() → str

Renders the heading as a markdown string. Markdown headings are returned using the # syntax where the number of # symbols corresponds to the heading level:

```
# This is an H1
## This is an H2
### This is an H3
```

Returns
the heading as a markdown string

demote() → *Heading*

Demotes a heading down a level. Fails silently if the heading is already at the lowest level (i.e., <h6>).

```
>>> heading = Heading("This is an H2 heading", 1).demote()
>>> str(heading)
'## This is an H2 heading'
```

Returns
self

`get_level()` → int

Retrieves the level of the heading.

```
>>> heading = Heading("This is the heading text", 1)
>>> heading.get_level()
1
```

New in version 2.2: Included to avoid protected member access scenarios.

Returns
the heading level

`get_text()` → str

Returns the heading text free of any styling. Useful when the heading is composed of various Inline elements, and the raw text is needed without styling or linking.

```
>>> heading = Heading("This is the heading text", 1)
>>> heading.get_text()
'This is the heading text'
```

Returns
the heading as a string

`promote()` → *Heading*

Promotes a heading up a level. Fails silently if the heading is already at the highest level (i.e., <h1>).

```
>>> heading = Heading("This is an H2 heading", 3).promote()
>>> str(heading)
'## This is an H2 heading'
```

Returns
self

HorizontalRule

`class snakemd.HorizontalRule`

Bases: *Block*

A horizontal rule is a line separating different sections of a document. Horizontal rules only come in one form, so there are no settings to adjust.

`__repr__()` → str

Renders self as an unambiguous string for development. In this case, it displays in the style of a dataclass, where instance variables are listed with their values.

```
>>> horizontal_rule = HorizontalRule()
>>> repr(horizontal_rule)
'HorizontalRule()'
```

Returns
the HorizontalRule object as a development string

__str__() → str

Renders the horizontal rule as a markdown string. Markdown horizontal rules come in a variety of flavors, but the format used in this repo is the triple asterisk (i.e., ***) to avoid clashes with list formatting.

Returns
the horizontal rule as a markdown string

MDList

class snakemd.**MDList**(*items: Iterable[str | Inline | Block]*, *ordered: bool = False*, *checked: None | bool | Iterable[bool] = None*)

Bases: *Block*

A markdown list is a standalone list that comes in three varieties: ordered, unordered, and checked.

Raises
ValueError – when the checked argument is an Iterable[bool] that does not match the number of top-level elements in the list

Parameters

- **items** (*Iterable[str | Inline | Block]*) – a "list" of objects to be rendered as a list
- **ordered** (*bool*) – the ordered state of the list
  - defaults to `False` which renders an unordered list (i.e., `-`)
  - set to `True` to render an ordered list (i.e., `1.`)
- **checked** (*None | bool | Iterable[bool]*) – the checked state of the list
  - defaults to `None` which excludes checkboxes from being rendered
  - set to `False` to render a series of unchecked boxes (i.e., `- [ ]`)
  - set to `True` to render a series of checked boxes (i.e., `- [x]`)
  - set to `Iterable[bool]` to render the checked status of the top-level list elements directly

__repr__() → str

Renders self as an unambiguous string for development. In this case, it displays in the style of a dataclass, where instance variables are listed with their values.

```
>>> mdlist = MDList(["Plus", "Ultra"])
>>> repr(mdlist)
"MDList(items=[Paragraph(...), Paragraph(...)],...)"
```

Returns
the MDList object as a development string

__str__() → str

Renders the markdown list as a markdown string. Markdown lists come in a variety of flavors and are customized according to the settings provided. For example, if the the ordered flag is set, an ordered list will be rendered in markdown. Unordered lists and checklists both use the hyphen syntax for markdown (i.e., `-`) to avoid clashes with horizontal rules:

```
- This is an unordered list item
- So, is this
```

Ordered lists use numbers for each list item:

```
1. This is an ordered list item
2. So, is this
```

Returns
the list as a markdown string

Paragraph

class snakemd.**Paragraph**(*content: str | Iterable[*Inline | *str]*)

Bases: *Block*

A paragraph is a standalone block of text.

Parameters
content (*str | Iterable[str | Inline]*) – the text to be rendered as a paragraph where whitespace is not respected (see *snakemd.Raw* for whitespace sensitive applications)

- set to a string to render a single line of unformatted text
- set to a "list" of text objects to render a paragraph with more granular control over the individual text objects (e.g., linking, styling, etc.)

__repr__() → str

Renders self as an unambiguous string for development. In this case, it displays in the style of a dataclass, where instance variables are listed with their values.

Like Heading, the actual format of the development string may be more complex than expected. Specifically, all of the contents are automatically converted to a list of Inline objects.

```
>>> paragraph = Paragraph("Howdy!")
>>> repr(paragraph)
"Paragraph(content=[Inline(text='Howdy!',...)])"
```

Returns
the Paragraph object as a development string

__str__() → str

Renders the paragraph as a markdown string. Markdown paragraphs are returned as a singular line of text with all of the underlying elements rendered as expected:

```
This is an example of a **paragraph** with _formatting_
```

Returns
the paragraph as a markdown string

add(*text: str | Inline*) → *Paragraph*

Adds a text object to the paragraph.

```
>>> paragraph = Paragraph("Hello! ").add("I come in peace")
>>> str(paragraph)
'Hello! I come in peace'
```

Parameters
text (*str | Inline*) – a custom Inline element

Returns
self

insert_link(*target: str*, *link: str*, *count: int = -1*) → *Paragraph*

A convenience method which inserts links in the paragraph for all matching instances of a target string. This method is modeled after `str.replace()`, so a count can be provided to limit the number of insertions. This method will not replace links of text that have already been linked. See *snakemd.Paragraph.replace_link()* for that behavior.

```
>>> paragraph = Paragraph("Go here for docs")
>>> paragraph.insert_link("here", "https://snakemd.io")
Paragraph(content=[...])
>>> str(paragraph)
'Go [here](https://snakemd.io) for docs'
```

Parameters

- **target** (*str*) – the string to link
- **link** (*str*) – the url or path
- **count** (*int*) – the number of links to insert; defaults to -1 (all)

Returns
self

replace(*target: str*, *replacement: str*, *count: int = -1*) → *Paragraph*

A convenience method which replaces a target string with a string of the users choice. Like *insert_link()*, this method is modeled after `str.replace()` of the standard library. As a result, a count can be provided to limit the number of strings replaced in the paragraph.

```
>>> paragraph = Paragraph("I come in piece").replace("piece", "peace")
>>> str(paragraph)
'I come in peace'
```

Parameters

- **target** (*str*) – the target string to replace
- **replacement** (*str*) – the string to insert in place of the target
- **count** (*int*) – the number of targets to replace; defaults to -1 (all)

Returns
self

replace_link(*target_link: str*, *replacement_link: str*, *count: int = -1*) → *Paragraph*

A convenience method which replaces matching URLs in the paragraph with a new url. Like *insert_link()* and *replace()*, this method is also modeled after `str.replace()`, so a count can be provided to limit the number of links replaced in the paragraph. This method is useful if you want to replace existing URLs but don't necessarily care what the anchor text is.

```
>>> old = "https://therenegadecoder.com"
>>> new = "https://snakemd.io"
>>> paragraph = Paragraph("Go here for docs")
>>> paragraph.insert_link("here", old).replace_link(old, new)
Paragraph(content=[...])
>>> str(paragraph)
'Go [here](https://snakemd.io) for docs'
```

Parameters

- **target_link** (*str*) – the link to replace
- **replacement_link** (*str*) – the link to swap in
- **count** (*int*) – the number of links to replace; defaults to -1 (all)

Returns
self

Quote

class snakemd.**Quote**(*content: str | Iterable[str | Inline | Block]*)

Bases: *Block*

A quote is a standalone block of emphasized text. Quotes can be nested and can contain other blocks.

Parameters
content (*str | Iterable[str | Inline | Block]*) – the text to be formatted as a Markdown quote

- set to a string to render a whitespace respected quote (similar to *snakemd.Code*)
- set to a "list" of text objects to render a document-like quote (i.e., all items will be separated by newlines)

__repr__() → str

The developer's string method to help make sense of objects. For the purposes of this repo, the __repr__ method should create strings that can be used to recreate the element, much like the built-in feature of dataclasses (a feature which may be adopted in future versions of snakemd). Ultimately, this method must be implemented by all inheriting classes.

Returns
an unambiguous representation of the element

__str__() → str

Renders the quote as a markdown string. Markdown quotes vary in syntax, but the general approach in this repo is to apply the quote symbol (i.e., >) to the front of each line in the quote:

```
> this
> is
> a quote
```

Quotes can also be nested. To make this possible, nested quotes are padded by empty quote lines:

```
> Outer quote
>
> > Inner quote
```

(continues on next page)

(continued from previous page)

```
>
> Outer quote
```

It's unclear what is the correct way to handle nested quotes, but this format seems to be the most friendly for GitHub markdown. Future work may involve including the option to removing the padding.

Returns
the quote formatted as a markdown string

Raw

class snakemd.**Raw**(*text: str*)

Bases: *Block*

Raw blocks allow a user to insert text into a Markdown document without any processing. Use this block to insert raw Markdown or other types of text (e.g., Jekyll frontmatter) into a document.

Parameters
text (*str*) – the raw text to append to a Document

__repr__() → str

The developer's string method to help make sense of objects. For the purposes of this repo, the __repr__ method should create strings that can be used to recreate the element, much like the built-in feature of dataclasses (a feature which may be adopted in future versions of snakemd). Ultimately, this method must be implemented by all inheriting classes.

Returns
an unambiguous representation of the element

__str__() → str

Renders the raw block as a markdown string. Raw markdown is unprocessed and passes directly through to the document.

Returns
the raw block as a markdown string

Table

class snakemd.**Table**(*header: Iterable[str | Inline | Paragraph]*, *body: Iterable[Iterable[str | Inline | Paragraph]] = None*, *align: None | Iterable[Align] = None*, *indent: int = 0*)

Bases: *Block*

A table is a standalone block of rows and columns. Data is rendered according to underlying Inline items.

Raises
ValueError –

- when rows of table are of varying lengths
- when lengths of header and rows of table do not match

Parameters

- **header** (*Iterable[str | Inline | Paragraph]*) – the header row of labels
- **body** (*Iterable[Iterable[str | Inline | Paragraph]]*) – the collection of rows of data; defaults to an empty list

- **align** (*None | Iterable[Align]*) – the column alignment; defaults to None
- **indent** (*int*) – indent size for the whole table; defaults to 0

class Align(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Align is an enum only used by the Table class to specify the alignment of various columns in the table.

CENTER = 3

LEFT = 1

RIGHT = 2

__repr__() → str

The developer's string method to help make sense of objects. For the purposes of this repo, the __repr__ method should create strings that can be used to recreate the element, much like the built-in feature of dataclasses (a feature which may be adopted in future versions of snakemd). Ultimately, this method must be implemented by all inheriting classes.

Returns
an unambiguous representation of the element

__str__() → str

Renders the table as a markdown string. Table markdown follows the standard pipe syntax:

```
| Header 1 | Header 2 |
| -------- | -------- |
| Item 1A  | Item 2A  |
| Item 1B  | Item 2B  |
```

Alignment code adds colons in the appropriate locations. Final tables are rendered according to the widest items in each column for readability.

Returns
a table as a markdown string

add_row(*row: Iterable[str | Inline | Paragraph]*) → *Table*

A convenience method which adds a row to the end of table. Use this method to build a table row-by-row rather than constructing the table rows upfront.

```
>>> table = Table(
... ["Rank", "Player"],
... [["1st", "Crosby"], ["2nd", "McDavid"]]
... )
>>> table.add_row(["3rd", "Matthews"])
Table(header=[...], body=[...], align=None, indent=0)
>>> print(table)
| Rank | Player   |
| ---- | -------- |
| 1st  | Crosby   |
| 2nd  | McDavid  |
| 3rd  | Matthews |
```

Raises
ValueError – when row is not the same width as the table header

Parameters
row (*Iterable[str | Inline | Paragraph]*) – a row of data

Returns
self

3.2.3 Inline Elements

One of the benefits of creating Block elements over using the Document methods is the control users now have over the underlying structure and style. Instead of being bound to the string inputs, users can provide Inline elements directly. For example, there is often a need to link Headings. This is not exactly possible through the Document methods as even the returned Heading object has no support for linking. However, with Inline elements, we can create a custom Heading to our standards:

```
>>> from snakemd import Heading, Inline, new_doc
>>> doc = new_doc()
>>> heading = doc.add_block(Heading(Inline("Hello, World!", "https://snakemd.io"), 2))
```

The remainder of this section introduces the Inline class.

Inline

class snakemd.**Inline**(*text: str*, *image: None | str = None*, *link: None | str = None*, *bold: bool = False*, *italics: bool = False*, *strikethrough: bool = False*, *code: bool = False*)

Bases: *Element*

The basic unit of text in markdown. All components which contain text are built using this class instead of strings directly. That way, those elements capture all styling information.

Inline element parameters are in order of precedence. In other words, image markdown is applied to the text first while code markdown is applied last. Due to this design, some forms of inline text are not possible. For example, inline elements can be used to show inline markdown as an inline code element (e.g., `![here](https://example.com)`). However, inline elements cannot be used to style inline code (e.g., `**`code`**`). If styled code is necessary, it's possible to render the inline element as a string and pass the result to another inline element.

Parameters

- **text** (*str*) – the inline text to render
- **image** (*None | str*) – the source (either url or path) associated with an image
  - defaults to `None`
  - set to a string representing a URL or path to render an image (i.e., `![text](image)`)
- **link** (*None | str*) – the link (either url or path) associated with the inline element
  - defaults to `None`
  - set to a string representing a URL or path to render a link (i.e., `[text](link)`)
- **bold** (*bool*) – the bold state of the inline text
  - defaults to `False`
  - set to `True` to render bold text (i.e., `**text**`)
- **italics** (*bool*) – the italics state of the inline element

- defaults to `False`
- set to `True` to render text in italics (i.e., `_text_`)

- **strikethrough** (*bool*) – the strikethrough state of the inline text
  - defaults to `False`
  - set to `True` to render text with a strikethrough (i.e., `~~text~~`)
- **code** (*bool*) – the code state of the inline text
  - defaults to `False`
  - set to `True` to render text as code (i.e., `` `text` ``)

__repr__() → str

Renders self as an unambiguous string for development. In this case, it displays in the style of a dataclass, where instance variables are listed with their values.

Returns
the Inline object as a development string

__str__() → str

Renders self as a markdown ready string. In this case, inline can represent many different types of data from stylized text to code, links, and images.

```
>>> inline = Inline("This is formatted text", bold=True, italics=True)
>>> str(inline)
'_**This is formatted text**_'
```

Returns
the Inline object as a markdown string

bold() → *Inline*

Adds bold styling to self.

```
>>> inline = Inline("This is bold text").bold()
>>> print(inline)
**This is bold text**
```

Returns
self

code() → *Inline*

Adds code style to self.

```
>>> inline = Inline("x = 5").code()
>>> print(inline)
`x = 5`
```

Returns
self

get_link() → str

Retrieves the link attribute of the Inline element.

```
>>> inline = Inline("Here", link="https://snakemd.io")
>>> inline.get_link()
'https://snakemd.io'
```

New in version 2.2: Included to avoid protected member access scenarios.

Returns
the link of the Inline element

get_text() → str

Retrieves the text attribute of the Inline element.

```
>>> inline = Inline("This is text")
>>> inline.get_text()
'This is text'
```

New in version 2.2: Included to avoid protected member access scenarios.

Returns
the text of the Inline element

is_link() → bool

Checks if the Inline object represents a link.

```
>>> inline = Inline("This is not a link")
>>> inline.is_link()
False
```

Returns
True if the object has a link; False otherwise

is_text() → bool

Checks if this Inline element is a text-only element. If not, it must be an image, a link, or a code snippet.

```
>>> inline = Inline("This is text")
>>> inline.is_text()
True
```

Returns
True if this is a text-only element; False otherwise

italicize() → *Inline*

Adds italics styling to self.

```
>>> inline = Inline("This is italicized text").italicize()
>>> print(inline)
_This is italicized text_
```

Returns
self

link(*link: str*) → *Inline*

Adds link to self.

```
>>> inline = Inline("here").link("https://snakemd.io")
>>> print(inline)
[here](https://snakemd.io)
```

Parameters
link (*str*) – the URL or path to apply to this Inline element

Returns
self

reset() → *Inline*

Removes all settings from self (e.g., bold, code, italics, url, etc.). All that will remain is the text itself.

```
>>> inline = Inline(
... "This is normal text",
... link="https://snakemd.io",
... bold=True
... )
>>> inline.reset()
Inline(text='This is normal text',...)
>>> print(inline)
This is normal text
```

Returns
self

strikethrough() → *Inline*

Adds strikethrough styling to self.

```
>>> inline = Inline("This is striked text").strikethrough()
>>> print(inline)
~~This is striked text~~
```

Returns
self

unbold() → *Inline*

Removes bold styling from self.

```
>>> inline = Inline("This is normal text", bold=True).unbold()
>>> print(inline)
This is normal text
```

Returns
self

uncode() → *Inline*

Removes code styling from self.

```
>>> inline = Inline("This is normal text", code=True).uncode()
>>> print(inline)
This is normal text
```

Returns
self

unitalicize() → *Inline*

Removes italics styling from self.

```
>>> inline = Inline("This is normal text", italics=True).unitalicize()
>>> print(inline)
This is normal text
```

Returns
self

unlink() → *Inline*

Removes link from self.

```
>>> inline = Inline("This is normal text", link="https://snakemd.io")
>>> inline.unlink()
Inline(text='This is normal text',... link=None,...)
>>> print(inline)
This is normal text
```

Returns
self

unstrikethrough() → *Inline*

Remove strikethrough styling from self.

```
>>> inline = Inline("This is normal text", strikethrough=True)
>>> inline.unstrikethrough()
Inline(text='This is normal text',... strikethrough=False,...)
>>> print(inline)
This is normal text
```

Returns
self

3.3 The Template API

While the document and element APIs are available for folks who are already somewhat familiar with Markdown, a template system is slowly being developed for folks who are looking for a bit more convenience. Ultimately, these folks can expect support for typical document sections such as tables of contents, footers, copyrights, and more.

3.3.1 Template Interface

To allow for templates to be integrated with documents seamlessly, the Template interface was developed to inherit directly from the Element interface, just like Block and Inline.

class snakemd.**Template**

Bases: *Element*

A template element in Markdown. A template can be thought of as a subdocument or collection of blocks. The entire purpose of the Template interface is to provide a superclass for a variety of abstractions over the typical markdown features. For example, Markdown has no feature for tables of contents, but a template could be created to generate one automatically for the user. In other words, templates are meant to be convience objects for our users.

One cool feature of templates is that they are lazy loaded. Unlike traditional elements, this means templates aren't fully loaded until they are about to be rendered. The benefit is that we can place templates in our documents as placeholders without much configuration. Then, right before the document is rendered, the template will be injected with a reference to the contents of the document. As a result, templates are able to take advantage of the final contents of the document, such as being able to generate a word count from the words in the document or generate a table of contents from the headings in the document.

Note that the user does not have to worry about lazy loading at all. The document will take care of the dependency injection. If, however, the user needs to render a template outside the context of a document, they must call the load function manually.

load(*elements: list[*snakemd.elements.Element*]*) → None

Loads the template with a list of elements, presumably from an existing document.

Parameters
elements – a list of document elements

3.3.2 Templates

The template library is humble but growing. Feel free to share your ideas for templates on the project page or Discord. If you'd like to help create templates, the interface is available for subclassing. Your templates can either be included directly in snakemd, or you're free to create your own template library by importing snakemd. In the former case, the template should make use of the Python standard library only and not make use of any external dependencies. In the latter case, that restriction does not apply. With that said, the existing templates can be found below.

CSVTable

class snakemd.**CSVTable**(*path: PathLike*, *encoding: str = 'utf-8'*)

Bases: *Template*

A CSV Table is a wrapper for the Table Block, which provides a seamless way to load CSV data into Markdown. Because Markdown tables are required to have headers, the first row of the CSV is assumed to be a header. Future iterations of this template may allow users to select the exact row for their header. Future iterations may also allow for different CSV dialects like Excel.

New in version 2.2: Included to showcase the possibilities of templates

Parameters

- **path** (*os.Pathlike*) – the path to a CSV file
- **encoding** (*str*) – the encoding of the CSV file; defaults to utf-8

TableOfContents

class snakemd.**TableOfContents**(*levels: range = range(2, 3)*)

Bases: *Template*

A Table of Contents is an element containing an ordered list of all the <*h2*> headings in the document by default. A range can be specified to customize which headings (e.g., <*h3*>) are included in the table of contents. This element can be placed anywhere in the document.

Changed in version 2.2: Removed the doc parameter

Parameters

levels (*range[int]*) – a range of integers representing the sequence of heading levels to include in the table of contents; defaults to range(2, 3)

CHAPTER

FOUR

RESOURCES

To help with all your SnakeMD needs, we've put together a set of resources organized by the major versions of the library. The lists themselves are sorted by publish date, with the most recently published resources first.

4.1 v2.x

- **2023-04-14**: How to Migrate to SnakeMD 2.0.0
- **2022-05-13**: The Complete Guide to SnakeMD: A Python Library for Generating Markdown

4.2 v0.x

- **2022-01-22**: SnakeMD 0.10.x Features Checklists
- **2021-09-24**: How to Generate Markdown in Python Using SnakeMD v0.x

CHAPTER

FIVE

VERSION HISTORY

Note: All versions of documentation are left in the condition in which they were generated. At times, the navigation may look different than expected.

In an effort to keep history of all the documentation for SnakeMD, we've included all old versions below as follows:

5.1 v2.x

- v2.2.0b1 [#140, #142, #143, #144, #145, #146, #149]
  - Expanded the Element requirements to include `__repr__()` for developer friendly strings
  - Reworked logging system to take advantage of lazy loading and new `__repr__()` methods
  - Expanded testing to verify the `__repr__()` strings can be used as Python code
  - Added a handful of getter methods to dissuade folks from using protected members of classes
  - Fixed jQuery issues in documentation
  - Incorporated linting (specifically pylint) in development workflow
  - Updated changelog string for consistency on PyPI
  - Introduced concept of lazy loading for templates to allow for processing of document contents at render time
  - Broke existing behavior of a handful of utilities:
    - Changed the `dir` parameter to `directory` for `dump()` method of `Document` to eliminate shadowing of built-in `dir`
    - Removed `doc` parameter of `TableOfContents` constructor in preference of new lazy loading system of templates
- v2.1.0 [#136]
  - Migrated build system from setup.py to poetry
- v2.0.0 [#131]
  - Setup code for 2023-04-14 release
  - See beta releases below for the bulk of the changes
- v2.0.0b2 [#129, #130]
  - Converted all code snippets in docs to doctests

  - Reworked string input for `Quote` to pass directly through raw
  - Updated language around parameters in documentation to provide a list of possible inputs and their effects
  - Replaced `url` parameter with `link` parameter in `insert_link()` method of `Paragraph`
- v2.0.0b1 [#104, #107, #108, #110, #113, #115, #118, #120, #122, #123, #125, #126]
  - Removed several deprecated items:
    - Classes
      - `MDCheckList`
      - `CheckBox`
      - `Verification`
    - Methods
      - `Document.add_element()`
      - `Document.add_header()`
      - `Document.check_for_errors()`
      - `Inline.verify_url()`
      - `Paragraph.verify_urls()`
      - `Paragaph.is_text()`
    - Parameters
      - `name` from `new_doc` and `Document`
      - `code` and `lang` from `Paragraph`
      - `quote` from `Paragaph`
      - `render()` and `verify()` from the entire repository
  - Replaced several deprecated items:
    - Classes
      - `Inline` replaces `InlineText`
      - `Heading` replaces `Header`
    - Methods
      - `Inline.is_link()` replaces `Inline.is_url()`
      - `Document.dump()` replaces `Document.output_page()`
    - Parameters
      - `link` replaces `url` in `Inline`
  - Added several new features:
    - Included a `Quote` block which allows for quote nesting
    - Incorporated `ValueError` exceptions in various class constructors
    - Started a resources page in documentation
    - Created a requirements file at the root of the repo to aid in development
  - Improved various aspects of the repo:

    * Expanded testing to 163 tests for 100% coverage
    * Clarified design of `Inline` to highlight precedence
    * Cleaned up documentation of pre-release version directives
    * Expanded types of inputs on various classes for quality of life
    * Changed behavior of horizontal rule to avoid clashes with list items
    * Fixed bugs in logs and expanded logging capabilities
    * Standardized docstring formatting
    * Updated README automation to use latest features

Note: The gap between v0.x and v2.x is not a mistake. Initial development of SnakeMD used v1.x versions, which contaminated the PyPI repository. To avoid failed releases due to version clashes, all v1.x versions have been deleted, and the project has jumped straight to v2.x. Consider v2.x to be the official release of the module. Anything prior to v2.x is considered a pre-release.

5.2 v0.x

- v0.15.0 [#97, #98, #99, #101]
  - Moved README generation code to repo root as a script
  - Expanded Heading constructor to support list of strings and Inline objects
  - Migrated code block support from Paragraph class into new Code class
- v0.14.0 [#84, #86, #89, #90, #91, #95]
  - Added Raw block for user formatted text
  - Replaced InlineText with Inline
  - Added Block and Inline classes
  - Deprecated MDCheckList and CheckBox
  - Replaced render with bulit-in str method
- v0.13.0 [#71, #74, #76, #78, #80, #82]
  - Created a replacement method for output_page called dump
  - Renamed Header class to Heading
  - Included deprecation warnings for both output_page and header as well as others affected
- v0.12.0 [#65, #66]
  - Added support for table generation on-the-fly (#64)
  - Reworked documentation to include proper headings and organization
  - Added support for strikethrough on InlineText elements (#58)
- v0.11.0 [#61, #62]
  - Added support for table indentation
- v0.10.1 [#59]

  - Enforced UTF-8 encoding in the output_page method (#54)
- v0.10.0 [#55, #56, #57]
  - Added the CheckBox class for creating checkboxes
  - Added the MDCheckList class for creating lists of checkboxes
  - Added a Document method for implementing easy checklists
  - Updated README to include a new section on checklists
- v0.9.3 [#50, #49]
  - Added multiple versions of Python testing
  - Restricted package to Python version 3.8+
  - Added Markdown linting for main README
- v0.9.0 [#47, #46, #45]
  - Added convenience function for creating new Document objects (#40)
  - Ported documentation to Read the Docs (#43)
- v0.8.1
  - Fixed an issue where nested lists did not render correctly
- v0.8.0
  - Added range feature to Table of Contents (#41)
- v0.7.0
  - Added replace_link() method to Paragraph
  - Added various state methods to InlineText
  - Expanded testing
  - Lowered log level to INFO for verify URL errors
  - Added code coverage to build
- v0.6.0
  - Restructured api, so snakemd is the import module
  - Updated usage page to show more features
  - Fixed issue where base docs link would reroute to index.html directly
- v0.5.0
  - Added favicon to docs (#26)
  - Added mass URL verification function to Paragraph class (#27)
  - Expanded testing to ensure code works as expected
  - Changed behavior of insert_link() to mimic str.replace() (#19)
  - Added a replace method to Paragraph (#27)
  - Added plausible tracking to latest version of docs (#25)
- v0.4.1
  - Added support for Python logging library (#22)

  - Expanded support for strings in the Header, Paragraph, and MDList classes
  - Fixed an issue where Paragraphs would sometimes render unexpected spaces (#23)
  - Added GitHub links to version history page
  - Added support for column alignment on tables (#4)
  - Fixed issue where tables sometimes wouldn't pretty print properly (#5)
- v0.3.0 [#21]
  - Gave documentation a major overhaul
  - Added support for paragraphs in MDList
  - Added is_text() method to Paragraph
  - Fixed issue where punctuation sometimes rendered with an extra space in front
- v0.2.0 [#17]
  - Added support for horizontal rules
  - Added automated testing through PyTest and GitHub Actions
  - Added document verification services
  - Added documentation link to README as well as info about installing the package
  - Fixed table of contents single render problem
  - Added a feature which allows users to insert links in existing paragraphs
- v0.1.0
  - Added support for links, lists, images, tables, code blocks, and quotes
  - Added a table of contents feature

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

G

H

I

L

M

N

P

Q

R

S

T

U