{Rendered by {Elsa Abbas} on {June 23, 2025}}

Minidown Parser

In this exercise, we'll build a miniature Markdown-like parser for a made-up language called "Minidown".

Minidown is a format that allows:

- Section headings,
- Paragraph text,
- Inline code,
- Displayed code, and
- Bulleted lists.

For this exercise, you are not permitted to use a tools that generate parsing code (e.g., parser generators like yacc, bison, or ANTLR), or libraries that deal with Markdown or Markdown-derivatives. If you please, you may use other libraries that may be helpful (e.g., regex, string-manipulation libraries). They are, however, not required for this exercise. See the section "Meta-requirements" for additional details.

Goal 1: Parsing Minidown

The first goal is to parse Minidown. Write a function

```
read_minidown_file(path)
```

which reads a Minidown file at the given path, and produces a representation of the Minidown of your choice. It will be useful to read the remaining goals to understand what we will want to do with this representation.

The following sections describe the format.

Documents

A Minidown file will be a sequence of section headings, paragraphs, bulleted lists, and displayed code blocks.

Paragraphs

A paragraph is almost any sequence of text, possibly containing inline code (see below for a description of inline code). Other components of a document are the only exceptions to what a paragraph is.

A single newline character does not break the sequence of text and is interpreted as a space.

Two newline characters delimit two paragraphs.

The following are five example paragraphs:

```
|Hello, world!
|
|This is a paragraph with the sentence
|broken into two lines.
|
| This is a paragraph with a space at the
|beginning.
|
| *This is a paragraph that was probably
|meant to be a bulleted list, but has
|incorrect formatting.
|
|This is a paragraph with `code` embedded.
```

Section headings

Section headings are represented by lines that begin with a `#` character, followed by a space, followed optionally by additional other text. The space is required.

Multple `#` can be used to denote subsections. There can be a maximum of six `#` characters to denote a sixth-level heading. Additional levels are interpreted as paragraph text.

These are three example section headings:

```
|# This is a first section heading
|
|## This is a subsection heading
|### This is a subsubsection heading; new newline needed
```

Note that the following would all render as elements of a paragraph

```
|#Missing leading space
|##
| ### The last heading also didn't have a space, and this heading has a preceding space.
|####### Headings can have up to 6 # marks
```

Inline code

Inline code can be a part of paragraphs, section headings, and bulleted lists. Inline code is any text written surrounded with backticks. The exception is that it is an error if the text contains a newline or the end-of-file is reached.

See the example in the section "Example of inline and displayed code".

Displayed code

If a line begins with a triple-backtick followed by a newline, then it begins a block of displayed code. Characters are read literally until another triple-backtick is found with a following newline.

Note that these rules around the triple-backtick are strict. If there is whitespace, or any other characters, before or after the triple-backtick, it is not considered a delimiter for a code block. So a file containing

```
| ```
|foo
|```x
```

would be an error, since it would be interpreted as three pairs of backticks, and the second pair contains a newline.

See the example in the section "Example of inline and displayed code".

Example of inline and displayed code

The following example contains a variety of instances of code:

- A section heading, which contains a piece of inline code.
- A paragraph, which contains no inline code.
- A block of displayed code
- Another paragraph, which contains inline code.

```
|# Declarations with `void` return values
|
|Consider the following example function declaration:
|
|```
|void f(char x);
|```
|
|It declares that `f` takes a `char` and returns nothing.
```

Bulleted lists

Bulleted lists are denoted by a single `* ` character in the leftmost column of the file followed by a space.

Each subsequent list element separated by a single newline. If there is a single newline but the next character isn't a `* `, then it is interpreted as a continuation of the previous bullet point.

Two newlines, or the end-of-file, ends the bulleted list.

If there are any characters on a line before the `* ` (i.e., it is not in the leftmost column), it is interpreted literally.

If there isn't a space immediately after the `* `, it is interpreted literally.

For example, the following is a valid bulleted list of two elements

```
|* Hello
|world!
|* Goodbye world!
| * (wait, this isn't a third bullet!)
|*and neither is this!
```

It renders as

- Hello world!
- Goodbye world! * (wait, this isn't a third bullet!) *and neither is this!

Goal 2: Render as HTML

Write a function

```
emit_minidown_as_html(x, output)
```

which writes an object `x` (produced by `read_minidown_file`) to `output `, which may be a stream and/or file. The HTML does not need to be formatted, minified, or pretty-printed. It is perfectly OK to use an HTML library for this part of the goal.

A document should be produced as

```
<html>
<body>
...
</body>
</html>
```

You may optionally include CSS via a `<style>` tag if preferred.

A heading with a `level` of `x` should be produced as

```
<hX> ... </hX>
```

A paragraph should be produced as

```
<p> ... </p>
```

A `displayed-code` element should be produced as

```
<pre>
...
</pre>
```

An `inline-code` element should be produced as

```
<code> ... </code>
```

A bulleted list should be produced as `<ul> ... </ul>` . Each bullet item should be produced as `<li> ... </li>` . So, for example, the bulleted list

```
|* A
|* B
|* C
```

should be produced as

```
<ul>
<li>A</li>
<li>B</li>
<li>C</li>
</ul>
```

In all HTML output, the following characters should be escaped everywhere:

- `&` becomes `&amp;`
- `<` becomes `&lt;`
- `>` becomes `&gt;`
- `"` becomes `&quot;`
- `'` becomes `&#39;`

Goal 3: A test document

Produce a test document that contains all of the syntactic features of Minidown. Include as many corner cases as you see reasonable. Add this file to your submission as `test.md` .

Produce HTML and render this in your browser. Print it as a PDF and submit it as `test.pdf` .

Goal 4: Render this document

Write

```
Rendered by {your first and last name} on {date}.
```

at the very top of this document, produce HTML, and render it in your browser. Print it as a PDF and submit it as `exercise.pdf` .

Meta-requirements

- Terminology: Above, we have used programming terms like "function," "type," "constructor," etc. These terms are being used broadly, and may be interpreted however you please in your programming language. For example, something specified above to be a "function" may be implemented in your programming language as a method, procedure, subroutine, or otherwise.
- Naming Conventions: You do not need to match the function and variable names exactly. Please feel free to choose whichever names you prefer and whichever names would be idiomatic in your programming language.
- Language: We prefer your code be written in Python.
- Libraries: Feel free to use any easily-accessible libraries that are utilitarian in nature. Do not use library functionality which directly solves any of the challenges. (For example, in this exercise, you're welcome to use an HTML library. However, you can't use or crib from or repurpose a Markdown parsing library.)
- Operating System: Your code should work on Windows or a UNIX system, like Linux or macOS. If this is not feasible for you, please let us know ahead of time.
- Quality: We know that this is a take-home exercise, and time constrains one's ability to make such a project fully robust. We ask that you code to your usual standard, but if shortcuts shall be made, to clearly mark them.
- Version Control: Do as you would in an ordinary, professional project. You may, if convenient, share a private GitHub repository with us.
- Testing: Do as you see fit, noting the flexibility of the "quality" requirement above.
- Documentation: Documentation should be done as you see fit, noting the flexibility of the "quality" requirement above.

The code you write is owned completely by you, and of course may be repurposed or used for anything you please. However, we humbly request that the code you write as it pertains to this exercise remains private so others have a fair chance.

Judgment criteria

You will be judged on overall software engineering quality, broad efficiency considerations (i.e., we are looking at big design choices, not micro-optimizations), and the extent to which your code satisfies the requirements outlined in this document.