reStructuredText Markup Specification

docutils.sourceforge.io/docs/ref/rst/restructuredtext.html

reStructuredText is plaintext that uses simple and intuitive constructs to indicate the structure of a document. These constructs are equally easy to read in raw and processed forms. This document is itself an example of reStructuredText (raw, if you are reading the text file, or processed, if you are reading an HTML document, for example). The reStructuredText parser is a component of Docutils.

Simple, implicit markup is used to indicate special constructs, such as section headings, bullet lists, and emphasis. The markup used is as minimal and unobtrusive as possible. Less often-used constructs and extensions to the basic reStructuredText syntax may have more elaborate or explicit markup.

reStructuredText is applicable to documents of any length, from the very small (such as inline program documentation fragments, e.g. Python docstrings) to the quite large (this document).

The first section gives a quick overview of the syntax of the reStructuredText markup by example. A complete specification is given in the Syntax Details section.

Literal blocks (in which no markup processing is done) are used for examples throughout this document, to illustrate the plaintext markup.

Quick Syntax Overview

A reStructuredText document is made up of body or block-level elements, and may be structured into sections. Sections are indicated through title style (underlines & optional overlines). Sections contain body elements and/or subsections. Some body elements contain further elements, such as lists containing list items, which in turn may contain paragraphs and other body elements. Others, such as paragraphs, contain text and inline markup elements.

Here are examples of body elements:

- Paragraphs (and inline markup):

```
Paragraphs contain text and may contain inline markup:
*emphasis*, **strong emphasis**, `interpreted text`, ``inline
literals``, standalone hyperlinks (https://www.python.org),
external hyperlinks (Python_), internal cross-references
(example_), footnote references ([1]_), citation references
([CIT2002]_), substitution references (|example|), and _`inline
internal targets`.

Paragraphs are separated by blank lines and are left-aligned.
```

- Five types of lists:

    1. Bullet lists:

        ```
        - This is a bullet list.

        - Bullets can be ``*``, ``+``, or ``-``.
        ```

    2. Enumerated lists:

        ```
        1. This is an enumerated list.

        2. Enumerators may be arabic numerals, letters, or roman
           numerals.
        ```

    3. Definition lists:

        ```
        what
            Definition lists associate a term with a definition.

        how
            The term is a one-line phrase, and the definition is one
            or more paragraphs or body elements, indented relative to
            the term.
        ```

    4. Field lists:

        ```
        :what: Field lists map field names to field bodies, like
               database records.  They are often part of an extension
               syntax.

        :how: The field marker is a colon, the field name, and a
              colon.

              The field body may contain one or more body elements,
              indented relative to the field marker.
        ```

    5. Option lists, for listing command-line options:

        ```
        -a            command-line option "a"
        -b file       options can have arguments
                      and long descriptions
        --long        options can be long also
        --input=file  long options can also have
                      arguments
        /V            DOS/VMS-style options too
        ```

        There must be at least two spaces between the option and the description.

- Literal blocks:

  ```
  Literal blocks are either indented or line-prefix-quoted blocks,
  and indicated with a double-colon (``::``) at the end of the
  preceding paragraph (right here -->)::

      if literal_block:
          text = 'is left as-is'
          spaces_and_linebreaks = 'are preserved'
          markup_processing = None
  ```

- Block quotes:

  ```
  Block quotes consist of indented body elements:

      This theory, that is mine, is mine.

      -- Anne Elk (Miss)
  ```

- Doctest blocks:

  ```
  >>> print 'Python-specific usage examples; begun with ">>> "'
  Python-specific usage examples; begun with ">>> "
  >>> print '(cut and pasted from interactive Python sessions)'
  (cut and pasted from interactive Python sessions)
  ```

- Two syntaxes for tables:

  1. Grid tables; complete, but complex and verbose:

     ```
     +------------------------+------------+----------+
     | Header row, column 1   | Header 2   | Header 3 |
     +========================+============+==========+
     | body row 1, column 1   | column 2   | column 3 |
     +------------------------+------------+----------+
     | body row 2             | Cells may span        |
     +------------------------+-----------------------+
     ```

  2. Simple tables; easy and compact, but limited:

     ```
     ====================  ==========  ==========
     Header row, column 1  Header 2    Header 3
     ====================  ==========  ==========
     body row 1, column 1  column 2    column 3
     body row 2            Cells may span columns
     ====================  ==========  ==========
     ```

- Explicit markup blocks all begin with an explicit block marker, two periods and a space:
  - Footnotes:

```
.. [1] A footnote contains body elements, consistently
 indented by at least 1 space.

    The least indented line sets the reference
    indentation, so this is a nested block quote.
```

  - Citations:

```
.. [CIT2002] Just like a footnote, except the label is
   textual.
```

  - Hyperlink targets:

```
.. _Python: https://www.python.org

.. _example:

The "_example" target above points to this paragraph.
```

  - Directives:

```
.. image:: mylogo.png
```

  - Substitution definitions:

```
.. |symbol here| image:: symbol.png
```

  - Comments:

```
.. Comments begin with two dots and whitespace.  Anything may
   follow, except for the syntax of footnotes/citations,
   hyperlink targets, directives, or substitution definitions.
```

Syntax Details

Descriptions below list "doctree elements" (document tree element names; XML DTD generic identifiers) corresponding to syntax constructs. For details on the hierarchy of elements, please see The Docutils Document Tree and the Docutils Generic DTD XML document type definition.

Whitespace

Config setting:
tab_width

Spaces are recommended for indentation, but tabs may also be used. Tabs will be converted to spaces. Tab stops are at every 8th column (processing systems may make this value configurable, Docutils uses the tab_width configuration setting).

Other whitespace characters (form feeds [chr(12)] and vertical tabs [chr(11)]) are converted to single spaces before processing.

Blank Lines

Blank lines are used to separate paragraphs and other elements. Multiple successive blank lines are equivalent to a single blank line, except within literal blocks (where all whitespace is preserved). Blank lines may be omitted when the markup makes element separation unambiguous, in conjunction with indentation. The first line of a document is treated as if it is preceded by a blank line, and the last line of a document is treated as if it is followed by a blank line.

Indentation

When a paragraph or other construct consists of more than one line of text, the lines must be left-aligned:

```
This is a paragraph.  The lines of
this paragraph are aligned at the left.

    This paragraph has problems.  The
lines are not left-aligned.  In addition
  to potential misinterpretation, warning
    and/or error messages will be generated
  by the parser.
```

Indentation is used to indicate -- and is only significant in indicating -- block quotes, definitions (in definition lists), and local nested content:

- list item content (multi-line contents of list items, and multiple body elements within a list item, including nested lists),
- the content of literal blocks, and
- the content of explicit markup blocks (directives, footnotes, ...).

Any text whose indentation is less than that of the current level (i.e., unindented text or "dedents") ends the current level of indentation.

Since all indentation is significant, the level of indentation must be consistent. For example, indentation is the sole markup indicator for block quotes:

```
This is a top-level paragraph.

    This paragraph belongs to a first-level block quote.

    Paragraph 2 of the first-level block quote.
```

Multiple levels of indentation within a block will result in more complex structures. The least indented line of a block sets the *reference indentation*:

```
This is a top-level paragraph.

    This paragraph belongs to a first-level block quote.

        This paragraph belongs to a second-level block quote.

Another top-level paragraph.

        This paragraph belongs to a second-level block quote.

    This paragraph belongs to a first-level block quote.
    The second-level block quote above is inside this
    first-level block quote.

Every block has its own reference indentation::

        This paragraph belongs to a first-level block quote
        because there is no less indented line in the block.
```

Several constructs begin with a marker, and the body of the construct must be indented relative to the marker. For constructs using simple markers (bullet lists, enumerated lists), the level of indentation of the body is determined by the position of the first line of text. For example:

```
- This is the first line of a bullet list
  item's paragraph.  All lines must align
  relative to the first line.

      This indented paragraph is interpreted
      as a block quote.

  Another paragraph belonging to the first list item.

 Because it is not sufficiently indented,
 this paragraph does not belong to the list
 item (it's a block quote following the list).
```

The first line of text may start below the marker:

```
1.
 This is the first line of an enumeration
 item's paragraph.

  This indented paragraph is interpreted
  as a block quote.

 This paragraph still belongs to the list
 item

This paragraph ends the list.
```

The body of explicit markup blocks, field lists, and option lists ends above the first line with the same or less indentation than the marker. For example, field lists may have very long markers (containing the field names):

```
:Hello: This field has a short field name, so aligning
        the field body with the first line is feasible.
:Long field name: It would be inconvenient to align the
    field body with the left edge of the first line.
:Number of African swallows required to carry a coconut:
    Sometimes, it is preferable to begin the body
    on the next line.
```

Escaping Mechanism

The character set universally available to plaintext documents, 7-bit ASCII, is limited. No matter what characters are used for markup, they will already have multiple meanings in written text. Therefore markup characters will sometimes appear in text without being intended as markup. Any serious markup system requires an escaping mechanism to override the default meaning of the characters used for the markup. In reStructuredText we use the *backslash*, commonly used as an escaping character in other domains.

A backslash (\) escapes the following character.

- "Escaping" backslash characters are represented by NULL characters in the Document Tree and removed from the output document by the Docutils writers.
- Escaped non-white characters are prevented from playing a role in any markup interpretation. The escaped character represents the character itself. (A literal backslash can be specified by two backslashes in a row -- the first backslash escapes the second. [1])
- Escaped whitespace characters are removed from the output document together with the escaping backslash. This allows for character-level inline markup.

  In *URI context* [2], backslash-escaped whitespace represents a single space.

Backslashes have no special meaning in *literal context* [3]. Here, a single backslash represents a literal backslash, without having to double up. [1]

Reference Names

Reference names identify elements for cross-referencing.

Simple reference names are single words consisting of alphanumerics plus isolated (no two adjacent) internal hyphens, underscores, periods, colons and plus signs; no whitespace or other characters are allowed. Footnote labels (Footnotes & Footnote References), citation labels (Citations & Citation References), interpreted text roles, and some hyperlink references use the simple reference name syntax.

Reference names using punctuation or whose names are phrases (two or more space-separated words) are called *phrase references*. Phrase-references are expressed by enclosing the phrase in backquotes and treating the backquoted text as a reference name:

```
Want to learn about `my favorite programming language`_?

.. _my favorite programming language: https://www.python.org
```

Simple reference names may also optionally use backquotes.

Reference names are whitespace-neutral and case-insensitive [4]: When resolving reference names internally,

- whitespace is normalized (one or more spaces, horizontal or vertical tabs, newlines, carriage returns, or form feeds, are interpreted as a single space), and
- case is normalized (all alphabetic characters are converted to lowercase). [4]

For example, the following hyperlink references are equivalent:

```
- `A HYPERLINK`_
- `a    hyperlink`_
- `A
  Hyperlink`_
```

Hyperlinks, footnotes, and citations all share the same namespace for reference names. [4] This means that a footnote defined as ".. [#note]" can be referred to by the footnote reference [#note]_ as well as a plain hyperlink reference note_. Of course, each type of reference (hyperlink, footnote, citation) may be processed and rendered differently. Some care should be taken to avoid reference name conflicts. [5]

Document Structure

Document

Doctree element:
<document>

The top-level element of a parsed reStructuredText document is the "document" element. After initial parsing, the document element is a simple container for a document fragment, consisting of body elements, transitions, and sections, but lacking a document title or other bibliographic elements. The code that calls the parser may choose to run one or more optional post-parse transforms, rearranging the document fragment into a complete document with a title and possibly other metadata elements (author, date, etc.; see Bibliographic Fields).

Document Title

Specifically, there is no special syntax to indicate a *document title* and *subtitle* in reStructuredText. [6] Instead, a uniquely-adorned top-level section title can be treated as the document title. Similarly, a uniquely-adorned second-level section title immediately after the document title can become the document subtitle. The rest of the sections are then lifted up a level or two. See A ReStructuredText Primer for examples and the DocTitle transform for details.

Sections

Doctree elements:
<section>, <title>

Sections are identified through their *titles*, which are marked up with adornment: "underlines" below the title text, or underlines and matching "overlines" above the title. An underline/overline is a single repeated punctuation character that begins in column 1 and forms a line extending at least as far as the right edge of the title text. [7] Specifically, an underline/overline character may be any non-alphanumeric printable 7-bit ASCII character [8]. When an overline is used, the length and character used must match the underline. Underline-only adornment styles are distinct from overline-and-underline styles that use the same character. There may be any number of levels of section titles, although some output formats may have limits (HTML has 6 levels).

Rather than imposing a fixed number and order of section title adornment styles, the order enforced will be the order as encountered. The first style encountered will be an outermost title (like HTML <H1>), the second style will be a subtitle, the third will be a subsubtitle, and so on.

Below are examples of section title styles:

```
===============
 Section Title
===============

---------------
 Section Title
---------------

Section Title
=============

Section Title
-------------

Section Title
^^^^^^^^^^^^^

Section Title
'''''''''''''

Section Title
.............

Section Title
~~~~~~~~~~~~~

Section Title
*************

Section Title
+++++++++++++

Section Title
^^^^^^^^^^^^^
```

When a title has both an underline and an overline, the title text may be inset, as in the first two examples above. This is merely aesthetic and not significant. Underline-only title text may *not* be inset.

A blank line after a title is optional. All text blocks up to the next title of the same or higher level are included in a section (or subsection, etc.).

All section title styles need not be used, nor need any specific section title style be used. However, a document must be consistent in its use of section titles: once a hierarchy of title styles is established, sections must use that hierarchy. [9]

Each section title automatically generates a hyperlink target pointing to the section. The text of the hyperlink target (the "reference name") is the same as that of the section title. See Implicit Hyperlink Targets for a complete description.

Sections may contain body elements, transitions, and nested sections.

Transitions

Doctree element:
<transition>

Instead of subheads, extra space or a type ornament between paragraphs may be used to mark text divisions or to signal changes in subject or emphasis.

(The Chicago Manual of Style, 14th edition, section 1.80)

Transitions are commonly seen in novels and short fiction, as a gap spanning one or more lines, with or without a type ornament such as a row of asterisks. Transitions separate other body elements. A transition should not begin or end a section or document, nor should two transitions be immediately adjacent.

The syntax for a transition marker is a horizontal line of 4 or more repeated punctuation characters. The syntax is the same as section title underlines without title text. Transition markers require blank lines before and after:

```
Para.

----------

Para.
```

Unlike section title underlines, no hierarchy of transition markers is enforced, nor do differences in transition markers accomplish anything. It is recommended that a single consistent style be used.

The processing system is free to render transitions in output in any way it likes. For example, horizontal rules (<hr>) in HTML output would be an obvious choice.

Body Elements

Paragraphs

Doctree element:
<paragraph>

Paragraphs consist of blocks of left-aligned text with no markup indicating any other body element. Blank lines separate paragraphs from each other and from other body elements. Paragraphs may contain inline markup.

Syntax diagram:

```
+------------------------------+
| paragraph                    |
|                              |
+------------------------------+

+------------------------------+
| paragraph                    |
|                              |
+------------------------------+
```

Bullet Lists

Doctree elements:
<bullet_list>, <list_item>

A text block which begins with a *, +, -, •, ‣, or ⁃, followed by whitespace, is a bullet list item (a.k.a. "unordered" list item). List item bodies must be left-aligned and indented relative to the bullet; the text immediately after the bullet determines the indentation. For example:

```
- This is the first bullet list item.  The blank line above the
  first list item is required; blank lines between list items
  (such as below this paragraph) are optional.

- This is the first paragraph in the second item in the list.

  This is the second paragraph in the second item in the list.
  The blank line above this paragraph is required.  The left edge
  of this paragraph lines up with the paragraph above, both
  indented relative to the bullet.

  - This is a sublist.  The bullet lines up with the left edge of
    the text blocks above.  A sublist is a new list so requires a
    blank line above and below.

- This is the third item of the main list.

This paragraph is not part of the list.
```

Here are examples of **incorrectly** formatted bullet lists:

```
- This first line is fine.
A blank line is required between list items and paragraphs.
(Warning)

- The following line appears to be a new sublist, but it is not:
  - This is a paragraph continuation, not a sublist (since there's
    no blank line).  This line is also incorrectly indented.
  - Warnings may be issued by the implementation.
```

Syntax diagram:

```
+------+-----------------------+
| "- " | list item             |
+------| (body elements)+      |
       +-----------------------+
```

Enumerated Lists

Doctree elements:
<enumerated_list>, <list_item>

Enumerated lists (a.k.a. "ordered" lists) are similar to bullet lists, but use enumerators instead of bullets. An enumerator consists of an enumeration sequence member and formatting, followed by whitespace. The following *enumeration sequences* are recognized:

- arabic numerals: 1, 2, 3, ... (no upper limit).
- uppercase alphabet characters: A, B, C, ..., Z.
- lower-case alphabet characters: a, b, c, ..., z.
- uppercase Roman numerals: I, II, III, IV, ..., MMMMCMXCIX (4999).
- lowercase Roman numerals: i, ii, iii, iv, ..., mmmmcmxcix (4999).

In addition, the auto-enumerator, #, may be used to automatically enumerate a list. Auto-enumerated lists may begin with explicit enumeration, which sets the sequence and start value. Fully auto-enumerated lists use arabic numerals and begin with 1.

The following *formatting types* are recognized:

- suffixed with a period: 1., A., a., I., i..
- surrounded by parentheses: (1), (A), (a), (I), (i).
- suffixed with a right-parenthesis: 1), A), a), I), i).

While parsing an enumerated list, a new list will be started whenever:

- An enumerator is encountered which does not have the same format and sequence type as the current list (e.g. 1., (a) produces two separate lists).
- The enumerators are not in sequence (e.g., 1., 3. produces two separate lists).

It is recommended that the enumerator of the first list item be ordinal-1 (1, A, a, I, or i). Although other start-values will be recognized, they may not be supported by the output format. A level-1 [info] system message will be generated for any list beginning with a non-ordinal-1 enumerator.

Lists using Roman numerals must begin with I/i or a multi-character value, such as II or XV. Any other single-character Roman numeral (V, X, L, C, D, M) will be interpreted as a letter of the alphabet, not as a Roman numeral. Likewise, lists using letters of the alphabet may not begin with I/i, since these are recognized as Roman numeral 1.

The second line of each enumerated list item is checked for validity. This is to prevent ordinary paragraphs from being mistakenly interpreted as list items, when they happen to begin with text identical to enumerators. For example, this text is parsed as an ordinary paragraph:

```
A. Einstein was a really
smart dude.
```

However, ambiguity cannot be avoided if the paragraph consists of only one line. This text is parsed as an enumerated list item:

```
A. Einstein was a really smart dude.
```

Examples of nested enumerated lists:

```
1. Item 1 initial text.

   a) Item 1a.
   b) Item 1b.

2. a) Item 2a.
   b) Item 2b.
```

Example syntax diagram:

```
+-------+----------------------+
| "1. " | list item            |
+-------| (body elements)+     |
        +----------------------+
```

Definition Lists

Doctree elements:
<definition_list>, <definition_list_item>, <term>, <classifier>, <definition>

Each definition list item contains a term, optional classifiers, and a definition.

- A *term* is a simple one-line word or phrase. Escape a leading hyphen to prevent recognition as an option list item.

- Optional *classifiers* may follow the term on the same line, each after an inline : (space, colon, space). Inline markup is parsed in the term line before the classifier delimiters are recognized. A delimiter will only be recognized if it appears outside of any inline markup.

- A *definition* is a block indented relative to the term, and may contain multiple paragraphs and other body elements. There may be no blank line between a term line and a definition block (this distinguishes definition lists from block quotes). Blank lines are required before the first and after the last definition list item, but are optional in-between.

Example:

```
term 1
    Definition 1.

term 2
    Definition 2, paragraph 1.

    Definition 2, paragraph 2.

term 3 : classifier
    Definition 3.

term 4 : classifier one : classifier two
    Definition 4.

\-term 5
    Without escaping, this would be an option list item.
```

A definition list may be used in various ways, including:

- As a dictionary or glossary. The term is the word itself, a classifier may be used to indicate the usage of the term (noun, verb, etc.), and the definition follows.

- To describe program variables. The term is the variable name, a classifier may be used to indicate the type of the variable (string, integer, etc.), and the definition describes the variable's use in the program. This usage of definition lists supports the classifier syntax of Grouch, a system for describing and enforcing a Python object schema.

Syntax diagram:

```
+----------------------------+
| term [ " : " classifier ]* |
+--+-------------------------+--+
   | definition                 |
   | (body elements)+           |
   +----------------------------+
```

Field Lists

Doctree elements:
<field_list>, <field>, <field_name>, <field_body>

Field lists are used as part of an extension syntax, such as options for directives, or database-like records meant for further processing. They may also be used for two-column table-like structures resembling database records (label & data pairs). Applications of reStructuredText may recognize field names and transform fields or field bodies in certain contexts. For examples, see Bibliographic Fields or directive options below or the "meta" directive.

Field lists are mappings from *field names* to *field bodies*, modeled on RFC822 headers. A field name may consist of any characters, but colons (:) inside of field names must be backslash-escaped when followed by whitespace.[10] Inline markup is parsed in field names, but care must be taken when using interpreted text with explicit roles in field names: the role must be a suffix to the interpreted text. Field names are case-insensitive when further processed or transformed. The field name, along with a single colon prefix and suffix, together form the field marker. The field marker is followed by whitespace and the field body. The field body may contain multiple body elements, indented relative to the field marker. The first line after the field name marker determines the indentation of the field body. For example:

```
:Date: 2001-08-16
:Version: 1
:Authors: - Me
          - Myself
          - I
:Indentation: Since the field marker may be quite long, the second
   and subsequent lines of the field body do not have to line up
   with the first line, but they must be indented relative to the
   field name marker, and they must line up with each other.
:Parameter i: integer
```

The interpretation of individual words in a multi-word field name is up to the application. The application may specify a syntax for the field name. For example, second and subsequent words may be treated as "arguments", quoted phrases may be treated as a single argument, and direct support for the "name=value" syntax may be added.

Standard RFC822 headers cannot be used for this construct because they are ambiguous. A word followed by a colon at the beginning of a line is common in written text. However, in well-defined contexts such as when a field list invariably occurs at the beginning of a document (PEPs and email messages), standard RFC822 headers could be used.

Syntax diagram (simplified):

```
+--------------------+----------------------+
| ":" field name ":" | field body           |
+-------+------------+                      |
        | (body elements)+                  |
        +-----------------------------------+
```

Bibliographic Fields

Doctree elements:
<docinfo>, <address>, <author>, <authors>, <contact>, <copyright>, <date>, <organization>, <revision>, <status>, <topic>, <version>

When a field list is the document body's first element [11], it may have its fields transformed to bibliographic data by the DocInfo transform. This bibliographic data corresponds to the front matter of a book, such as the title page and copyright page.

Certain registered field names (listed below) are recognized and transformed to the corresponding doctree elements, most becoming child elements of the <docinfo> element. No ordering is required of these fields, although they may be rearranged to fit the document structure, as noted. Unless otherwise indicated below, each of the bibliographic elements' field bodies may contain a single paragraph only. Field bodies may be checked for RCS keywords and cleaned up. Any unrecognized fields will remain as generic fields in the docinfo element.

The registered bibliographic field names and their corresponding doctree elements are as follows:

field name [12]	doctree element
Abstract	<topic>
Address	<address>
Author	<author>
Authors	<authors>
Contact	<contact>
Copyright	<copyright>
Date	<date>
Dedication	<topic>
Organization	<organization>
Revision	<revision>
Status	<status>
Version	<version>

The **Authors** field may contain

- a single paragraph consisting of a list of authors, separated by ; or , [12] (the semicolon is checked first, so Doe, Jane; Doe, John will work),
- multiple paragraphs (one per author), or
- a bullet list whose elements each contain a single paragraph per author.

As a convention, use an "Authors" field for authors with common affiliation and separate "Author" fields (each followed by the respective "Organization", "Address", and/or "Contact" fields) for authors with distinct affiliations. There is currently no way to represent the organization or contact info of an individual author in an "Authors" field.

In some languages (e.g. Swedish), there is no singular/plural distinction between "Author" and "Authors", so only an "Authors" field is provided, and a single name is interpreted as an "Author". If a single name contains a comma, end it with a semicolon or use a one-item bullet list to disambiguate:

```
:Författare: * Larsson, Lars
```

The **Address** field is for a multi-line surface mailing address. Newlines will be preserved.

The **Dedication** and **Abstract** fields may contain arbitrary body elements. Only one of each is allowed. They become <topic> elements with "Dedication" or "Abstract" titles (or language equivalents) [12] immediately following the <docinfo> element.

Unregistered/generic fields may contain one or more paragraphs or arbitrary body elements. To support custom styling, the field name is also added to the "classes" attribute value after an identifier normalization.

RCS Keywords

Bibliographic fields recognized by the parser are normally checked for RCS [13] keywords and cleaned up [14]. RCS keywords may be entered into source files as "$keyword$", and once stored under RCS, CVS [15], or SVN [16], they are expanded to "$keyword: expansion text $". For example, a "Status" field will be transformed to a "status" element:

```
:Status: $keyword: expansion text $
```

Processed, the "status" element's text will become simply "expansion text". The dollar sign delimiters and leading RCS keyword name are removed.

The RCS keyword processing only kicks in when the field list is in bibliographic context (first non-comment construct in the document, after a document title if there is one).

Option Lists

Doctree elements:
<option_list>, <option_list_item>, <option_group>, <option>, <option_string>, <option_argument>, <description>

Option lists map a program's command-line options to descriptions documenting them. For example:

```
-a         Output all.
-c arg     Output just arg.
--long     Output all day long.
/V         A VMS/DOS-style option.

-p         This option has two paragraphs in the description.
           This is the first.

           This is the second.
           Blank lines may be omitted between options
           (as above) or left in (as here and below).

--very-long-option  A VMS-style option.  Note the adjustment
                    for the required two spaces.

--an-even-longer-option
           The description can also start on the next line.

-2, --two  This option has two variants.

-f FILE, --file=FILE  These two options are synonyms; both have
                      arguments.

-f <[path]file>  Option argument placeholders must start with
                 a letter or be wrapped in angle brackets.

-d <src dest>    Angle brackets are also required if an option
                 expects more than one argument.
```

There are several types of options recognized by reStructuredText:

- Short POSIX options consist of one dash and an option letter.
- Long POSIX options consist of two dashes and an option word; some systems use a single dash.
- Old GNU-style "plus" options consist of one plus and an option letter ("plus" options are deprecated now, their use discouraged).
- DOS/VMS options consist of a slash and an option letter or word.

Please note that both POSIX-style and DOS/VMS-style options may be used by DOS or Windows software. These and other variations are sometimes used mixed together. The names above have been chosen for convenience only.

The syntax for short and long POSIX options is based on the syntax supported by Python's getopt.py module, which implements an option parser similar to the GNU libc getopt_long() function but with some restrictions. There are many variant option systems, and reStructuredText option lists do not support all of them.

Although long POSIX and DOS/VMS option words may be allowed to be truncated by the operating system or the application when used on the command line, reStructuredText option lists do not show or support this with any special syntax. The complete option word should be given, supported by notes about truncation if and when applicable.

Options may be followed by an argument placeholder, whose role and syntax should be explained in the description text. Either a space or an equals sign may be used as a delimiter between long options and option argument placeholders; short options (- or + prefix only) use a space or omit the delimiter. Option arguments may take one of two forms:

- Begins with a letter ([a-zA-Z]) and subsequently consists of letters, numbers, underscores and hyphens ([a-zA-Z0-9_-]).
- Begins with an open-angle-bracket (<) and ends with a close-angle-bracket (>); any characters except angle brackets are allowed internally.

Multiple option "synonyms" may be listed, sharing a single description. They must be separated by comma-space.

There must be at least two spaces between the option(s) and the description (which can also start on the next line). The description may contain multiple body elements. The first line after the option marker determines the indentation of the description. As with other types of lists, blank lines are required before the first option list item and after the last, but are optional between option entries.

Syntax diagram (simplified):

```
+-----------------------------+-------------+
| option [" " argument] "  " | description |
+-------+--------------------+             |
        | (body elements)+                 |
        +----------------------------------+
```

Literal Blocks

Doctree element:
<literal_block>

A paragraph consisting of two colons (::) signifies that the following text block(s) comprise a literal block. The literal block must either be indented or quoted (see below). No markup processing is done within a literal block. It is left as-is, and is typically rendered in a monospaced typeface:

```
This is a typical paragraph.  An indented literal block follows.

::

    for a in [5,4,3,2,1]:   # this is program code, shown as-is
        print a
    print "it's..."
    # a literal block continues until the indentation ends

This text has returned to the indentation of the first paragraph,
is outside of the literal block, and is therefore treated as an
ordinary paragraph.
```

The paragraph containing only :: will be completely removed from the output; no empty paragraph will remain.

As a convenience, the :: is also recognized at the end of any paragraph. If immediately preceded by whitespace, both colons will be removed from the output (this is the "partially minimized" form). When text immediately precedes the ::, *one* colon will be removed from the output, leaving only one colon visible (i.e., :: will be replaced by :; this is the "fully minimized" form).

In other words, these are all equivalent (please pay attention to the colons after "Paragraph"):

1. Expanded form:

   ```
   Paragraph:

   ::

       Literal block
   ```

2. Partially minimized form:

   ```
   Paragraph: ::

       Literal block
   ```

3. Fully minimized form:

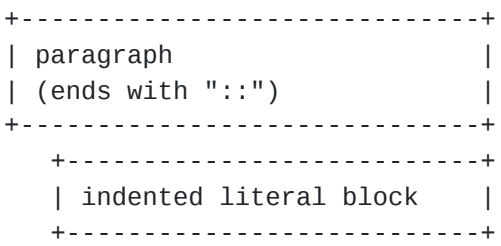
   ```
   Paragraph::

       Literal block
   ```

All whitespace (including line breaks, but excluding minimum indentation for indented literal blocks) is preserved. Blank lines are required before and after a literal block, but these blank lines are not included as part of the literal block.

Indented Literal Blocks

Indented literal blocks are indicated by indentation relative to the surrounding text (leading whitespace on each line). The minimum indentation will be removed from each line of an indented literal block. The literal block need not be contiguous; blank lines are allowed between sections of indented text. The literal block ends with the end of the indentation.

Syntax diagram:



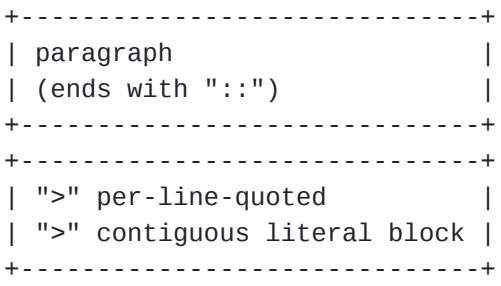
Quoted Literal Blocks

Quoted literal blocks are unindented contiguous blocks of text where each line begins with the same non-alphanumeric printable 7-bit ASCII character [17]. A blank line ends a quoted literal block. The quoting characters are preserved in the processed document.

Possible uses include literate programming in Haskell and email quoting:

```
John Doe wrote::

>> Great idea!
>
> Why didn't I think of that?

You just did!  ;-)
```

Syntax diagram:



Line Blocks

Doctree elements:
<line_block>, <line>

Line blocks are useful for address blocks, verse (poetry, song lyrics), and unadorned lists, where the structure of lines is significant. Line blocks are groups of lines beginning with vertical bar (|) prefixes. Each vertical bar prefix indicates a new line, so line breaks are preserved. Initial indents are also significant, resulting in a nested structure. Inline markup is supported. Continuation lines are wrapped portions of long lines; they begin with a space in place of the vertical bar. The left edge of a continuation line must be indented, but need not be aligned with the left edge of the text above it. A line block ends with a blank line.

This example illustrates continuation lines:

```
| Lend us a couple of bob till Thursday.
| I'm absolutely skint.
| But I'm expecting a postal order and I can pay you back
  as soon as it comes.
| Love, Ewan.
```

This example illustrates the nesting of line blocks, indicated by the initial indentation of new lines:

```
Take it away, Eric the Orchestra Leader!

    | A one, two, a one two three four
    |
    | Half a bee, philosophically,
    |     must, *ipso facto*, half not be.
    | But half the bee has got to be,
    |     *vis a vis* its entity.  D'you see?
    |
    | But can a bee be said to be
    |     or not to be an entire bee,
    |         when half the bee is not a bee,
    |             due to some ancient injury?
    |
    | Singing...
```

Syntax diagram:

```
+------+-----------------------+
| "| " | line                  |
+------| continuation line     |
       +-----------------------+
```

Block Quotes

Doctree elements:
<block_quote>, <attribution>

A text block that is indented relative to the preceding text, without preceding markup indicating it to be a literal block or other content, is a block quote. All markup processing (for body elements and inline markup) continues within the block quote:

```
This is an ordinary paragraph, introducing a block quote.

    "It is my business to know things.  That is my trade."

    -- Sherlock Holmes
```

A block quote may end with an attribution: a paragraph beginning with --, ---, or a true em-dash, flush left within the block quote. If the attribution consists of multiple lines, the left edges of the second and subsequent lines must align.

Multiple block quotes may occur consecutively if terminated with attributions.

> Unindented paragraph.
>
> > Block quote 1.
> >
> > —Attribution 1
>
> > Block quote 2.

Empty comments may be used to explicitly terminate preceding constructs that would otherwise consume a block quote:

```
* List item.

..

    Block quote 3.
```

Empty comments may also be used to separate block quotes:

```
    Block quote 4.

..

    Block quote 5.
```

Blank lines are required before and after a block quote, but these blank lines are not included as part of the block quote.

Syntax diagram:

```
+-----------------------------+
| (current level of           |
| indentation)                |
+-----------------------------+
   +---------------------------+
   | block quote               |
   | (body elements)+          |
   |                           |
   | -- attribution text       |
   |    (optional)             |
   +---------------------------+
```

Doctest Blocks

Doctree element:
<doctest_block>

Doctest blocks are interactive Python sessions cut-and-pasted into docstrings. They are meant to illustrate usage by example, and provide an elegant and powerful testing environment via the doctest module in the Python standard library.

Doctest blocks are text blocks which begin with the Python interactive interpreter main prompt (>>> followed by a space) and end with a blank line. Doctest blocks are treated as a special case of literal blocks, without requiring the literal block syntax. If both are present, the literal block syntax takes priority over Doctest block syntax:

```
This is an ordinary paragraph.

>>> print 'this is a Doctest block'
this is a Doctest block

The following is a literal block::

    >>> This is not recognized as a doctest block by
    reStructuredText.  It *will* be recognized by the doctest
    module, though!
```

Indentation is not required for doctest blocks.

Tables

Doctree elements:
<table>, <tgroup>, <colspec>, <thead>, <tbody>, <row>, <entry>

ReStructuredText provides two syntax variants for delineating table cells: Grid Tables and Simple Tables. Tables are also generated by the "csv-table" and "list-table" directives. The "table" directive is used to add a table title, caption, or specify options.

As with other body elements, blank lines are required before and after tables. Tables' left edges should align with the left edge of preceding text blocks; if indented, the table is considered to be part of a block quote.

Once isolated, each table cell is treated as a miniature document; the top and bottom cell boundaries act as delimiting blank lines. Each cell contains zero or more body elements. Cell contents may include left and/or right margins, which are removed before processing.

Grid Tables

Grid tables provide a complete table representation via grid-like "ASCII art". Grid tables allow arbitrary cell contents (body elements), and both row and column spans. However, grid tables can be cumbersome to produce, especially for simple data sets. The Emacs table mode is a tool that allows easy editing of grid tables, in Emacs. See Simple Tables for a simpler (but limited) representation.

Grid tables are described with a visual grid made up of the characters -, =, |, and +. The hyphen (-) is used for horizontal lines (row separators). The equals sign (=) may be used to separate optional header rows from the table body (not supported by the Emacs table mode). The vertical bar (|) is used for vertical lines (column separators). The plus sign (+) is used for intersections of horizontal and vertical lines. Example:

```
+------------------------+------------+----------+----------+
| Header row, column 1   | Header 2   | Header 3 | Header 4 |
| (header rows optional) |            |          |          |
+========================+============+==========+==========+
| body row 1, column 1   | column 2   | column 3 | column 4 |
+------------------------+------------+----------+----------+
| body row 2             | Cells may span columns.          |
+------------------------+------------+---------------------+
| body row 3             | Cells may  | - Table cells       |
+------------------------+ span rows. | - contain           |
| body row 4             |            | - body elements.    |
+------------------------+------------+---------------------+
```

Some care must be taken with grid tables to avoid undesired interactions with cell text in rare cases. For example, the following table contains a cell in row 2 spanning from column 2 to column 4:

```
+--------------+----------+-----------+-----------+
| row 1, col 1 | column 2 | column 3  | column 4  |
+--------------+----------+-----------+-----------+
| row 2        |                                  |
+--------------+----------+-----------+-----------+
| row 3        |          |           |           |
+--------------+----------+-----------+-----------+
```

If a vertical bar is used in the text of that cell, it could have unintended effects if accidentally aligned with column boundaries:

```
+--------------+----------+-----------+-----------+
| row 1, col 1 | column 2 | column 3  | column 4  |
+--------------+----------+-----------+-----------+
| row 2        | Use the command ``ls | more``.   |
+--------------+----------+-----------+-----------+
| row 3        |          |           |           |
+--------------+----------+-----------+-----------+
```

Several solutions are possible. All that is needed is to break the continuity of the cell outline rectangle. One possibility is to shift the text by adding an extra space before:

```
+--------------+----------+-----------+-----------+
| row 1, col 1 | column 2 | column 3  | column 4  |
+--------------+----------+-----------+-----------+
| row 2        |  Use the command ``ls | more``.  |
+--------------+----------+-----------+-----------+
| row 3        |          |           |           |
+--------------+----------+-----------+-----------+
```

Another possibility is to add an extra line to row 2:

```
+--------------+----------+-----------+-----------+
| row 1, col 1 | column 2 | column 3  | column 4  |
+--------------+----------+-----------+-----------+
| row 2        | Use the command ``ls | more``.   |
|              |                                  |
+--------------+----------+-----------+-----------+
| row 3        |          |           |           |
+--------------+----------+-----------+-----------+
```

Simple Tables

Simple tables provide a compact and easy to type but limited row-oriented table representation for simple data sets. Cell contents are typically single paragraphs, although arbitrary body elements may be represented in most cells. Simple tables allow multi-line rows (in all but the first column) and column spans, but not row spans. See Grid Tables above for a complete table representation.

Simple tables are described with horizontal borders made up of = and - characters. The equals sign (=) is used for top and bottom table borders, and to separate optional header rows from the table body. The hyphen (-) is used to indicate column spans in a single row by underlining the joined columns, and may optionally be used to explicitly and/or visually separate rows.

A simple table begins with a top border of equals signs with one or more spaces at each column boundary (two or more spaces recommended). Regardless of spans, the top border *must* fully describe all table columns. There must be at least two columns in the table (to

differentiate it from section headers). The top border may be followed by header rows, and the last of the optional header rows is underlined with =, again with spaces at column boundaries. There may not be a blank line below the header row separator; it would be interpreted as the bottom border of the table. The bottom boundary of the table consists of = underlines, also with spaces at column boundaries. For example, here is a truth table, a three-column table with one header row and four body rows:

```
=====  =====  =======
  A      B    A and B
=====  =====  =======
False  False  False
True   False  False
False  True   False
True   True   True
=====  =====  =======
```

Underlines of - may be used to indicate column spans by "filling in" column margins to join adjacent columns. Column span underlines must be complete (they must cover all columns) and align with established column boundaries. Text lines containing column span underlines may not contain any other text. A column span underline applies only to one row immediately above it. For example, here is a table with a column span in the header:

```
=====  =====  ======
   Inputs     Output
------------  ------
  A      B    A or B
=====  =====  ======
False  False  False
True   False  True
False  True   True
True   True   True
=====  =====  ======
```

Each line of text must contain spaces at column boundaries, except where cells have been joined by column spans. Each line of text starts a new row, except when there is a blank cell in the first column. In that case, that line of text is parsed as a continuation line. For this reason, cells in the first column of new rows (*not* continuation lines) *must* contain some text; blank cells would lead to a misinterpretation (but see the tip below). Also, this mechanism limits cells in the first column to only one line of text. Use grid tables if this limitation is unacceptable.

Underlines of - may also be used to visually separate rows, even if there are no column spans. This is especially useful in long tables, where rows are many lines long.

Blank lines are permitted within simple tables. Their interpretation depends on the context. Blank lines *between* rows are ignored. Blank lines *within* multi-line rows may separate paragraphs or other body elements within cells.

The rightmost column is unbounded; text may continue past the edge of the table (as indicated by the table borders). However, it is recommended that borders be made long enough to contain the entire text.

The following example illustrates continuation lines (row 2 consists of two lines of text, and four lines for row 3), a blank line separating paragraphs (row 3, column 2), text extending past the right edge of the table, and a new row which will have no text in the first column in the processed output (row 4):

```
=====  =====
col 1  col 2
=====  =====
1      Second column of row 1.
2      Second column of row 2.
       Second line of paragraph.
3      - Second column of row 3.

       - Second item in bullet
         list (row 3, column 2).
\      Row 4; column 1 will be empty.
=====  =====
```

Explicit Markup Blocks

The explicit markup syntax is used for footnotes, citations, hyperlink targets, directives, substitution definitions, and comments.

An explicit markup block is a text block:

- whose first line begins with .. followed by whitespace (the *explicit markup start*),
- whose second and subsequent lines (if any) are indented relative to the first, and
- which ends before an unindented line.

Explicit markup blocks are analogous to field list items. The maximum common indentation is always removed from the second and subsequent lines of the block body. Therefore, if the first construct fits in one line and the indentation of the first and second constructs should differ, the first construct should not begin on the same line as the explicit markup start.

Blank lines are required between explicit markup blocks and other elements, but are optional between explicit markup blocks where unambiguous.

Citations

Doctree element:
<citation>

See also:
citation references

Citations are identical to manually numbered footnotes except that they use non-numeric labels such as [note] or [GVR2001]. Citation labels are simple reference names (case-insensitive single words consisting of alphanumerics plus internal hyphens, underscores, and periods; no whitespace). Citations may be rendered separately and differently from footnotes. For example:

```
Here is a citation reference: [CIT2002]_.

.. [CIT2002] This is the citation.  It's just like a footnote,
   except the label is textual.
```

Hyperlink Targets

Doctree element:
<target>

See also:
hyperlink references
embedded URIs and aliases
inline internal targets

These are also called *explicit hyperlink targets*, to differentiate them from implicit hyperlink targets defined below.

Hyperlink targets identify a location within or outside of a document, which may be linked to by hyperlink references.

Hyperlink targets may be named or anonymous. *Named hyperlink targets* consist of an explicit markup start (..), an underscore, the reference name (no trailing underscore), a colon, whitespace, and a link block:

```
.. _hyperlink-name: link-block
```

Reference names are whitespace-neutral and case-insensitive. See Reference Names for details and examples.

Anonymous hyperlink targets consist of an explicit markup start (..), two underscores, a colon, whitespace, and a link block; there is no reference name:

```
.. __: anonymous-hyperlink-target-link-block
```

An alternate syntax for anonymous hyperlinks consists of two underscores, a space, and a link block:

```
__ anonymous-hyperlink-target-link-block
```

See Anonymous Hyperlinks below.

There are three types of hyperlink targets: internal, external, and indirect.

1. Internal hyperlink targets have empty link blocks. They provide an end point allowing a hyperlink to connect one place to another within a document. An internal hyperlink target points to the element following the target. [19] For example:

   ```
   Clicking on this internal hyperlink will take us to the target_
   below.

   .. _target:

   The hyperlink target above points to this paragraph.
   ```

   Internal hyperlink targets may be "chained". Multiple adjacent internal hyperlink targets all point to the same element:

   ```
   .. _target1:
   .. _target2:

   The targets "target1" and "target2" are synonyms; they both
   point to this paragraph.
   ```

   If the element "pointed to" is an external hyperlink target (with a URI in its link block; see #2 below) the URI from the external hyperlink target is propagated to the internal hyperlink targets; they will all "point to" the same URI. There is no need to duplicate a URI. For example, all three of the following hyperlink targets refer to the same URI:

   ```
   .. _Python DOC-SIG mailing list archive:
   .. _archive:
   .. _Doc-SIG: https://mail.python.org/pipermail/doc-sig/
   ```

   An inline form of internal hyperlink target is available; see Inline Internal Targets.

2. External hyperlink targets have a URI-reference or email address in their link blocks. For example, take the following input:

```
See the Python_ home page for info.

`Write to me`_ with your questions.

.. _Python: https://www.python.org
.. _Write to me: jdoe@example.com
```

After processing into HTML, the hyperlinks might be expressed as:

```
See the <a href="https://www.python.org">Python</a> home page
for info.

<a href="mailto:jdoe@example.com">Write to me</a> with your
questions.
```

An external hyperlink's URI may begin on the same line as the explicit markup start and target name, or it may begin in an indented text block immediately following, with no intervening blank lines. If there are multiple lines in the link block, they are concatenated. Any unescaped whitespace is removed (whitespace is permitted to allow for line wrapping). The following external hyperlink targets are equivalent:

```
.. _one-liner: https://docutils.sourceforge.io/rst.html

.. _starts-on-this-line: https://
   docutils.sourceforge.net/rst.html

.. _entirely-below:
   https://docutils.
   sourceforge.net/rst.html
```

Escaped whitespace is preserved as intentional spaces, e.g.:

```
.. _reference: ../local\ path\ with\ spaces.html
```

If an external hyperlink target's URI contains an underscore as its last character, it must be escaped to avoid being mistaken for an indirect hyperlink target:

```
This link_ refers to a file called ``underscore_``.

.. _link: underscore\_
```

It is possible (although not generally recommended) to include URIs directly within hyperlink references. See Embedded URIs and Aliases below.

3. Indirect hyperlink targets have a hyperlink reference in their link blocks. In the following example, target "one" indirectly references whatever target "two" references, and target "two" references target "three", an internal hyperlink target. In effect, all three reference the same thing:

   ```
   .. _one: two_
   .. _two: three_
   .. _three:
   ```

   Just as with hyperlink references anywhere else in a document, if a phrase-reference is used in the link block it must be enclosed in backquotes. As with external hyperlink targets, the link block of an indirect hyperlink target may begin on the same line as the explicit markup start or the next line. It may also be split over multiple lines, in which case the lines are joined with whitespace before being normalized.

   For example, the following indirect hyperlink targets are equivalent:

   ```
   .. _one-liner: `A HYPERLINK`_
   .. _entirely-below:
      `a    hyperlink`_
   .. _split: `A
      Hyperlink`_
   ```

   It is possible to include an alias directly within hyperlink references. See Embedded URIs and Aliases below.

If the reference name contains any colons, either:

- the phrase must be enclosed in backquotes:

  ```
  .. _`FAQTS: Computers: Programming: Languages: Python`:
     http://python.faqts.com/
  ```

- or the colon(s) must be backslash-escaped in the link target:

  ```
  .. _Chapter One\: "Tadpole Days":

  It's not easy being green...
  ```

See Implicit Hyperlink Targets below for the resolution of duplicate reference names.

Syntax diagram:

```
+-------+----------------------+
| ".. " | "_" name ":" link    |
+-------+ block                |
        |                      |
        +----------------------+
```

Anonymous Hyperlinks

The World Wide Web Consortium recommends in its HTML Techniques for Web Content Accessibility Guidelines that authors should "clearly identify the target of each link." Hyperlink references should be as verbose as possible, but duplicating a verbose hyperlink name in the target is onerous and error-prone. *Anonymous hyperlinks* are designed to allow convenient verbose hyperlink references, and are analogous to auto-numbered footnotes. They are particularly useful in short or one-off documents. However, this feature is easily abused and can result in unreadable plaintext and/or unmaintainable documents. Caution is advised.

Anonymous hyperlink references are specified with two underscores instead of one:

```
See `the web site of my favorite programming language`__.
```

Anonymous hyperlink targets begin with .. __:, no reference name is required or allowed:

```
.. __: https://www.python.org
```

As a convenient alternative, anonymous targets may begin with two underscores only:

```
__ https://www.python.org
```

The order of anonymous hyperlink references and targets within the document is significant: the first anonymous reference will link to the first anonymous target. The number of anonymous hyperlink references in a document must match the number of anonymous targets.

For readability, it is recommended that targets be kept close to references. Take care when editing text containing anonymous references; adding, removing, and rearranging references require attention to the order of corresponding targets.

Directives

Doctree elements:
depend on the directive

Directives are an extension mechanism for reStructuredText, a way of adding support for new constructs without adding new primary syntax (directives may support additional syntax locally). All standard directives (those implemented and registered in the reference reStructuredText parser) are described in the reStructuredText Directives document, and are always available. Any other directives are domain-specific, and may require special action to make them available when processing the document.

For example, the "image" directive is used to include an image:

```
.. image:: mylogo.jpeg
```

A graphic with a caption may be included with the "figure" directive:

```
.. figure:: larch.png

   The larch.
```

An "admonition" (note, caution, etc.) contains other body elements:

```
.. note:: This is a paragraph

   - Here is a bullet list.
```

Directives are indicated by an explicit markup start (..) followed by the directive type, two colons, and whitespace (together called the *directive marker*). Directive types are case-insensitive single words (alphanumerics plus isolated internal hyphens, underscores, plus signs, colons, and periods; no whitespace). Two colons are used after the directive type for these reasons:

- Two colons are distinctive, and unlikely to be used in common text.
- Two colons avoids clashes with common comment text like:

  ```
  .. Danger: modify at your own risk!
  ```

- If an implementation of reStructuredText does not recognize a directive (i.e., the directive-handler is not installed), a level-3 (error) system message is generated, and the entire directive block (including the directive itself) will be included as a literal block. Thus :: is a natural choice.

The directive block consists of any text on the first line of the directive after the directive marker, and any subsequent indented text. The interpretation of the directive block is up to the directive code. There are three logical parts to the directive block:

1. Directive arguments.
2. Directive options.
3. Directive content.

Individual directives can employ any combination of these parts. Directive arguments can be filesystem paths, URLs, title text, etc. Directive options are indicated using field lists; the field names and contents are directive-specific. Arguments and options must form a contiguous block beginning on the first or second line of the directive; a blank line indicates the beginning of the directive content block. If either arguments and/or options are employed by the directive, a blank line must separate them from the directive content. The "figure" directive employs all three parts:

```
.. figure:: larch.png
   :scale: 50

   The larch.
```

Simple directives may not require any content. If a directive that does not employ a content block is followed by indented text anyway, it is an error. If a block quote should immediately follow a directive, use an empty comment in-between (see Comments below).

Actions taken in response to directives and the interpretation of text in the directive content block or subsequent text block(s) are directive-dependent. See reStructuredText Directives for details.

Directives are meant for the arbitrary processing of their contents, which can be transformed into something possibly unrelated to the original text. It may also be possible for directives to be used as pragmas, to modify the behavior of the parser, such as to experiment with alternate syntax. There is no parser support for this functionality at present; if a reasonable need for pragma directives is found, they may be supported.

Directives do not generate "directive" elements; they are a *parser construct* only, and have no intrinsic meaning outside of reStructuredText. Instead, the parser will transform recognized directives into (possibly specialized) document elements. Unknown directives will trigger level-3 (error) system messages.

Syntax diagram:

```
+-------+-------------------------------+
| ".. " | directive type "::" directive |
+-------+ block                         |
        |                               |
        +-------------------------------+
```

Substitution Definitions

Doctree element:
<substitution_definition>

See also:
substitution references

Substitution definitions are indicated by an explicit markup start (..) followed by a vertical bar, the substitution text, another vertical bar, whitespace, and the definition block. Substitution text may not begin or end with whitespace. A substitution definition block contains an embedded inline-compatible directive (such as "image" or "replace") without the leading dots. For example:

```
The |biohazard| symbol must be used on containers used to
dispose of medical waste.

.. |biohazard| image:: biohazard.png
```

It is an error for a substitution definition block to directly or indirectly contain a circular substitution reference.

Substitution references are replaced in-line by the processed contents of the corresponding definition (linked by matching substitution text). Matches are case-sensitive but forgiving; if no exact match is found, a case-insensitive comparison is attempted.

Substitution definitions allow the power and flexibility of block-level directives to be shared by inline text. They are a way to include arbitrarily complex inline structures within text, while keeping the details out of the flow of text. They are the equivalent of SGML/XML's named entities or programming language macros.

Without the substitution mechanism, every time someone wants an application-specific new inline structure, they would have to petition for a syntax change. In combination with existing directive syntax, any inline structure can be coded without new syntax (except possibly a new directive).

Syntax diagram:

```
+-------+----------------------------------------------------+
| ".. " | "|" substitution text "| " directive type "::" data |
+-------+ directive block                                     |
        |                                                    |
        +----------------------------------------------------+
```

The following *inline-compatible directives* are implemented in Docutils:

"date":
inserts the current local date.

"image":
can be used for block-level images as well as in a substitution definition for inline images.

"raw":
can be used in block-level context as well as in a substitution definition.

"replace":
allows simple macro substitution. It also provides a workaround for the still missing support of nested inline markup.

"unicode":
converts Unicode character codes to characters.

Applications may find other use cases for the substitution mechanism. The following are ideas that have not been implemented in Docutils.

Objects
Substitution references may be used to associate ambiguous text with a unique object identifier.

For example, many sites may wish to implement an inline "user" directive:

```
|Michael| and |Jon| are our widget-wranglers.

.. |Michael| user:: mjones
.. |Jon|     user:: jhl
```

Depending on the needs of the site, this may be used to index the document for later searching, to hyperlink the inline text in various ways (mailto, homepage, mouseover Javascript with profile and contact information, etc.), or to customize presentation of the text (include username in the inline text, include an icon image with a link next to the text, make the text bold or a different color, etc.).

The same approach can be used in documents which frequently refer to a particular type of objects with unique identifiers but ambiguous common names. Movies, albums, books, photos, court cases, and laws are possible. For example:

```
|The Transparent Society| offers a fascinating alternate view
on privacy issues.

.. |The Transparent Society| book:: isbn=0738201448
```

Classes or functions, in contexts where the module or class names are unclear and/or interpreted text cannot be used, are another possibility:

```
4XSLT has the convenience method |runString|, so you don't
have to mess with DOM objects if all you want is the
transformed output.

.. |runString| function:: module=xml.xslt class=Processor
```

Templates
Inline markup may be used for later processing by a template engine. For example, a Zope author might write:

```
Welcome back, |name|!

.. |name| tal:: replace user/getUserName
```

After processing, this ZPT output would result:

```
Welcome back,
<span tal:replace="user/getUserName">name</span>!
```

Zope would then transform this to something like "Welcome back, David!" during a session with an actual user.

Comments

Doctree element:
<comment>

Config setting:
strip_comments

Explicit markup blocks that are not recognized as citations, directives, footnotes, hyperlink targets, or substitution definitions will be processed as a comment element.

Arbitrary indented text may be used on the lines following the explicit markup start:

```
.. This is a comment
..
   _so: is this!
..
   [and] this!
..
   this:: too!
..
   |even| this:: !

.. [this] however, is a citation.
```

Apart from removing the maximum common indentation, no further processing is done on the content; a comment contains a single "text blob". Depending on the output formatter, comments may be removed from the processed output. In Docutils, the strip_comments configuration setting triggers the removal of comment elements from the document tree.

Syntax diagram:

```
+-------+----------------------+
| ".. " | comment              |
+-------+ block                |
        |                      |
        +----------------------+
```

Empty Comments

An explicit markup start followed by a blank line and nothing else (apart from whitespace) is an "empty comment". It serves to terminate a preceding construct, and does **not** consume any indented text following. To have a block quote follow a list or any indented construct, insert an unindented empty comment in-between:

```
This is
  a definition list.

..

  This is a block quote.
```

Inline Markup

In reStructuredText, inline markup applies to words or phrases within a text block. The same whitespace and punctuation that serves to delimit words in written text is used to delimit the inline markup syntax constructs (see the inline markup recognition rules for details). The text within inline markup may not begin or end with whitespace. Arbitrary character-level inline markup is supported although not encouraged. Inline markup cannot be nested.

There are nine inline markup constructs. Five of the constructs use identical start-strings and end-strings to indicate the markup:

- emphasis: *
- strong emphasis: **
- interpreted text: `
- inline literals: ``
- substitution references: |

Three constructs use different start-strings and end-strings:

- inline internal targets: _` and `
- footnote references: [and]_
- hyperlink references: ` and `_ (phrases), or just a trailing _ (single words)

Standalone hyperlinks are recognized implicitly, and use no extra markup.

Inline comments are not supported.

Inline markup recognition rules

Config setting:
character_level_inline_markup

Inline markup start-strings and end-strings are only recognized if the following conditions are met:

1. Inline markup start-strings must be immediately followed by non-whitespace.
2. Inline markup end-strings must be immediately preceded by non-whitespace.
3. The inline markup end-string must be separated by at least one character from the start-string.
4. Both, inline markup start-string and end-string must not be preceded by an unescaped backslash (except for the end-string of inline literals). See Escaping Mechanism above for details.
5. If an inline markup start-string is immediately preceded by one of the ASCII characters ' " < ([{ or a similar non-ASCII character [20], it must not be followed by the corresponding closing character from ' " >)] } or a similar non-ASCII character [21]. (For quotes, matching characters can be any of the quotation marks in international usage.)

If the configuration setting character_level_inline_markup is False (default), additional conditions apply to the characters "around" the inline markup:

6. Inline markup start-strings must start a text block or be immediately preceded by
    - whitespace,
    - one of the ASCII characters - : / ' " < ([{
    - or a similar non-ASCII punctuation character. [22]
7. Inline markup end-strings must end a text block or be immediately followed by
    - whitespace,
    - one of the ASCII characters - . , : ; ! ? \ / ' ")] } >
    - or a similar non-ASCII punctuation character. [23]

The inline markup recognition rules were devised to allow 90% of non-markup uses of *, `, _, and | without escaping. For example, none of the following terms are recognized as containing inline markup strings:

- 2 * x a ** b (* BOM32_* ` `` _ __ | (breaks rule 1)
- || (breaks rule 3)
- "*" '|' (*) [*] {*} <*> ‘*’ ,*‘ ‘*, ’*’ ,*’ “*” „*“ “*„ ”*” „*” »*« ›*‹ «*» »*» ›*› (breaks rule 5)
- 2*x a**b O(N**2) e**(x*y) f(x)*f(y) a|b file*.* __init__ __init__() (breaks rule 6)

No escaping is required inside the following inline markup examples:

- *2 * x *a **b *.rst* (breaks rule 2; renders as "*2 * x *a **b *.rst*")
- *2*x a**b O(N**2) e**(x*y) f(x)*f(y) a*(1+2)* (breaks rule 7; renders as "*2*x a**b O(N**2) e**(x*y) f(x)*f(y) a*(1+2)*")

It may be desirable to use inline literals for some of these anyhow, especially if they represent code snippets. It's a judgment call.

The following terms *do* require either literal-quoting or escaping to avoid misinterpretation:

```
*4, class_, *args, **kwargs, `TeX-quoted', *ML, *.rst
```

In most use cases, inline literals or literal blocks are the best choice (by default, this also selects a monospaced font). Alternatively, the inline markup characters can be escaped:

```
\*4, class\_, \*args, \**kwargs, \`TeX-quoted', \*ML, \*.rst
```

For languages that don't use whitespace between words (e.g. Japanese or Chinese) it is recommended to set character_level_inline_markup to True and escape inline markup characters in case of non-markup use. The examples breaking rules 6 and 7 above show which constructs may need special attention.

Recognition order

Inline markup delimiter characters are used for multiple constructs, so to avoid ambiguity there must be a specific recognition order for each character. The inline markup recognition order is as follows:

- Asterisks: Strong emphasis (**) is recognized before emphasis (*).
- Backquotes: Inline literals (``), inline internal targets (leading _`, trailing `), are mutually independent, and are recognized before phrase hyperlink references (leading `, trailing `_) and interpreted text (`).
- Trailing underscores: Footnote references ([+ label +]_) and simple hyperlink references (name + trailing _) are mutually independent.
- Vertical bars: Substitution references (|) are independently recognized.
- Standalone hyperlinks are the last to be recognized.

Character-Level Inline Markup

Config setting:
character_level_inline_markup

It is possible to mark up individual characters within a word with backslash escapes (see Escaping Mechanism above). Backslash escapes can be used to allow arbitrary text to immediately follow inline markup:

```
Python ``list``\s use square bracket syntax.
```

The backslash will disappear from the processed document. The word "list" will appear as inline literal text, and the letter "s" will immediately follow it as normal text, with no space in-between.

Arbitrary text may immediately precede inline markup using backslash-escaped whitespace:

```
Possible in *re*\ ``Structured``\ *Text*, though not encouraged.
```

The backslashes and spaces separating "re", "Structured", and "Text" above will disappear from the processed document.

In Docutils, you may relax the inline markup recognition rules with the character_level_inline_markup setting, however this may in turn lead to false positives.

Emphasis

Doctree element:
<emphasis>

Start/End string:
*

Standard role:
:emphasis:

Text enclosed by single asterisk characters is emphasized:

```
This is *emphasized text*.
```

Emphasized text is typically displayed in italics.

Strong Emphasis

Doctree element:

Start/End string:
**

Standard role:
:strong:

Text enclosed by double-asterisks is emphasized strongly:

```
This is **strong text**.
```

Strongly emphasized text is typically displayed in boldface.

Interpreted Text

Doctree element:
depends on the explicit or implicit role and processing

Start/End string:
`

Configuration:
"default-role" directive

See also:
reStructuredText Interpreted Text Roles

Interpreted text is text that is meant to be related, indexed, linked, summarized, or otherwise processed, but the text itself is typically left alone. Interpreted text is enclosed by single backquote characters:

```
This is `interpreted text`.
```

The *role* of the interpreted text determines how the text is interpreted. The role may be inferred implicitly (as above; the *default role* is used) or indicated explicitly, using a role marker. A role marker consists of a colon, the role name, and another colon. A role name is a single word consisting of alphanumerics plus isolated internal hyphens, underscores, plus signs, colons, and periods; no whitespace or other characters are allowed. A role marker is either a prefix or a suffix to the interpreted text, whichever reads better; it's up to the author:

```
:role:`interpreted text`
```

```
`interpreted text`:role:
```

Interpreted text allows extensions to the available inline descriptive markup constructs. To emphasis, strong emphasis, inline literals, and hyperlink references, we can add "title-reference", "index-entry", "acronym", "class", "red", "blinking" or anything else we want (as long as it is a simple reference name). Only pre-determined roles are recognized; unknown roles will generate errors.

A core set of *standard roles* is implemented in the reference parser; see reStructuredText Interpreted Text Roles for individual descriptions. The "role" directive can be used to define custom interpreted text roles. In addition, applications may support specialized roles. The

default role (used for interpreted text without role marker) can be set with the "default-role" directive.

In field lists, care must be taken when using interpreted text with explicit roles in field names: the role must be a suffix to the interpreted text. The following are recognized as field list items:

```
:`field name`:code:: interpreted text with explicit role as suffix

:a `complex`:code:\  field name: a backslash-escaped space
                                 is necessary
```

The following are **not** recognized as field list items:

```
::code:`not a field name`: paragraph with interpreted text

:\ :code:`not a field name`: paragraph with interpreted text
```

Edge cases:

```
:field\:`name`: interpreted text (standard role) requires
                escaping the leading colon in a field name

:field:\`name`: not interpreted text
```

Inline Literals

Doctree element:
<literal>

Start/End string:
``

Standard role:
:literal:

See also:
:code:

Text enclosed by double-backquotes is treated as inline literals:

```
This text is an example of ``inline literals``.
```

Inline literals may contain any characters except two adjacent backquotes in an end-string context (according to the recognition rules above). No markup interpretation (including backslash-escape interpretation) is done within inline literals.

Line breaks and sequences of whitespace characters are *not* protected in inline literals. Although a reStructuredText parser will preserve them in its output, the final representation of the processed document depends on the output formatter, thus the preservation of whitespace cannot be guaranteed. If the preservation of line breaks and/or other whitespace is important, literal blocks should be used.

Inline literals or the :code: role are useful for short code snippets. For example:

```
The regular expression ``[+-]?(\d+(\.\d*)?|\.\d+)`` matches
floating-point numbers (without exponents).
```

Hyperlink References

Doctree element:
<reference>

Start/End strings:

reference type	name	start	end
named	simple	none	_
	phrase	`	`_
anonymous	simple	none	__
	phrase	`	`__

See also:
hyperlink targets

Hyperlink references are indicated by a trailing underscore (_) except for standalone hyperlinks which are recognized independently. The underscore can be thought of as a right-pointing arrow. The trailing underscores point away from hyperlink references, and the leading underscores point toward hyperlink targets.

Hyperlinks consist of two parts:

1. In the text body, there is a *reference*, a reference name with a trailing underscore (or two underscores for anonymous hyperlinks):

   ```
   See the Python_ home page for info.
   ```

2. A matching *target* must exist in the document. It may be embedded (see below) or exist somewhere else in the document (explicit hyperlink targets, implicit hyperlink targets, inline internal targets, directives with "name" option, citations, or numbered footnotes).

Named hyperlinks match reference and target by their reference names. *Anonymous* hyperlinks match references to targets by their order within the document.

Embedded URIs and Aliases

Doctree elements:
<reference>, <target>

Start/End strings:
< > (only recognized inside hyperlink references)

A hyperlink reference may directly embed a target URI or an "alias" hyperlink reference within angle brackets as follows:

```
See the `Python home page <https://www.python.org>`_ for info.

This `link <Python home page_>`_ is an alias to the link above.
```

This is equivalent to:

```
See the `Python home page`_ for info.

This link_ is an alias to the link above.

.. _Python home page: https://www.python.org
.. _link: `Python home page`_
```

The bracketed URI must be preceded by whitespace and be the last text before the end string.

With a single trailing underscore, the reference is *named* and generates an implicit hyperlink target.

With two trailing underscores, the reference is anonymous and the link text cannot be re-used as reference name. These are "one-off" hyperlinks. For example:

```
`RFC 2396 <https://www.rfc-editor.org/rfc/rfc2396.txt>`__ and `RFC
2732 <https://www.rfc-editor.org/rfc/rfc2732.txt>`__ together
define the syntax of URIs.
```

is equivalent to:

```
`RFC 2396`__ and `RFC 2732`__ together define the syntax of URIs.

__ https://www.rfc-editor.org/rfc/rfc2396.txt
__ https://www.rfc-editor.org/rfc/rfc2732.txt
```

Standalone hyperlinks are treated as URIs, even if they end with an underscore like in the example of a Python function documentation:

```
`__init__ <http:example.py.html#__init__>`__
```

If a target URI that is not recognized as standalone hyperlink happens to end with an underscore, this needs to be backslash-escaped to avoid being parsed as hyperlink reference. For example

```
Use the `source <parrots.rst\_>`__.
```

creates an anonymous reference to the file parrots.rst_.

If the reference text happens to end with angle-bracketed text that is *not* a URI or hyperlink reference, at least one angle-bracket needs to be backslash-escaped or an escaped space should follow. For example, here are three references to titles describing a tag:

```
See `HTML Element: \<a>`_, `HTML Element: <b\>`_, and
`HTML Element: <c>\ `_.
```

The reference text may also be omitted, in which case the URI will be duplicated for use as the reference text. This is useful for URI-references where the address or file name is also the desired reference text:

```
See `<a_named_relative_link>`_ or
`<an_anonymous_relative_link>`__ for details.
```

Inline Internal Targets

Doctree element:
<target>

Start/End strings:
_` `
_

See also:
hyperlink targets

Inline internal targets are the equivalent of explicit internal hyperlink targets, but may appear within running text. The syntax begins with an underscore and a backquote, is followed by a hyperlink name or phrase, and ends with a backquote. Inline internal targets may not be anonymous.

For example, the following paragraph contains a hyperlink target named "Norwegian Blue":

```
Oh yes, the _`Norwegian Blue`.  What's, um, what's wrong with it?
```

See Implicit Hyperlink Targets for the resolution of duplicate reference names.

Citation References

Doctree element:
<citation_reference>

Start/End string:
[ ]_

See also:
citations

Each citation reference consists of a square-bracketed label followed by a trailing underscore. Citation labels are simple reference names.

For example:

```
Here is a citation reference: [CIT2002]_.
```

Substitution References

Doctree elements:
<substitution_reference>, <reference>

Start/End string:
| (optionally followed by _ or __).

See also:
substitution definitions

Vertical bars are used to bracket the substitution reference text. A substitution reference may also be a hyperlink reference by appending a _ (named) or __ (anonymous) suffix; the substitution text is used for the reference text in the named case.

The processing system replaces substitution references with the processed contents of the corresponding substitution definitions (which see for the definition of "correspond"). Substitution definitions produce inline-compatible elements.

Examples:

```
This is a simple |substitution reference|.  It will be
replaced by the processing system.

This is a combination |substitution and hyperlink reference|_.
In addition to being replaced, the replacement text or element
will refer to the "substitution and hyperlink reference" target.
```

Standalone Hyperlinks

Doctree element:
<reference>

Start/End string:
none

A URI [25] or standalone email address within a text block is treated as a general external hyperlink with the URI rsp. address itself as the link's text. For example:

```
See https://www.python.org.
```

would be marked up in HTML as:

```
See <a href="https://www.python.org">https://www.python.org</a>.
```

Two forms of standalone hyperlinks are recognized:

1. URIs with known schemes (per the Official IANA Registry of URI Schemes and the W3C's Retired Index of WWW Addressing Schemes). Examples:

   ```
   tel:+48-816-555-1212
   urn:isbn:0-486-27557-4
   news:comp.infosystems.www.servers.unix
   https://john.doe@www.example.com:123/forum/questions/?
   tag=networking&order=newest#top
   ```

   With queries, fragments, and %-escape sequences, URIs can become quite complicated. A reStructuredText parser must be able to recognize any URI, as defined in RFC3936.

2. Standalone email addresses, which are treated as if they were URIs with a "mailto:" scheme. Example:

   ```
   someone@somewhere.com
   ```

Punctuation at the end of a URI is not considered part of the URI, unless the URI is terminated by a closing angle bracket (>). Backslashes may be used in URIs to escape markup characters, specifically asterisks (*) and underscores (_) which are valid URI characters (see Escaping Mechanism above).

Implicit Hyperlink Targets

Doctree elements:
<section>, <target>

Implicit hyperlink targets are generated by section titles and named hyperlink references with embedded URIs and aliases. They may also be generated by extension constructs. Implicit hyperlink targets behave identically to explicit hyperlink targets except in case of duplicate reference names.

Ambiguity due to different objects with the same reference name is avoided by the following procedure:

1. Duplicate external or indirect hyperlink targets that refer to the same URI or hyperlink reference do not conflict. One target is invalidated and an INFO [26] system message inserted.

2. Explicit hyperlink targets, citations, numbered footnotes, and directives with "name" option override any implicit targets having the same reference name. The implicit hyperlink target is invalidated, and an INFO [26] system message inserted.

3. Duplicate implicit hyperlink targets are both invalidated, and INFO [26] system messages inserted. For example, if two or more sections have the same title (such as "Introduction" subsections of a rigidly-structured document), there will be duplicate implicit hyperlink targets.

4. Duplicate explicit hyperlink targets (or other objects with the same reference name) are both invalidated, and WARNING [26] system messages inserted.

The parser returns a set of *unique* hyperlink targets. The calling software (such as Docutils) can warn of unresolvable links, giving reasons for the messages.

Measures and Units

Measures consist of a positive floating point number in standard (non-scientific) notation and an optional unit, possibly separated by one or more spaces.

Measures are only supported where explicitly mentioned in the reference manuals (directive option values of type "length" or "percentage"). In the document tree, they are stored in attributes of type measure.

It is up to the processing system to provide a fallback/workaround or raise an error if the output format does not support a unit (or values without unit). For the behaviour of the Docutils writers, see the writer documentation.

Length Units

The reStructuredText parser supports the length units in CSS3. [27] Unit identifiers are case-sensitive (in contrast to CSS):

em	the element's font size	
ex	x-height of the element's font	
ch	width of the “0” (ZERO, U+0030) glyph in the element’s font	
rem	font size of the root element	
vw	1% of the viewport (or paper) width	
vh	1% of the viewport (or paper) height	
vmin	1% of the viewport’s smaller dimension	
vmax	1% of the viewport’s larger dimension	
cm	centimeters	1 cm = 10 mm
mm	millimeters	1 mm = 1/1000 m
Q	quarter-millimeters	1 Q = 1/4 mm [28]
in	inches	1 in = 25.4 mm = 96 px
pc	picas	1 pc = 1/6 in = 12 pt
pt	points	1 pt = 1/72 in
px	pixels	1 px = 3/4 pt = 1/96 in [29]

The following are all valid length values: 1.5em, 20 mm, .5 in, 42.

Percentage Unit

Percentage values have a percent sign (%) as unit. Percentage values are relative to other values, depending on the context in which they occur.

Error Handling

Doctree elements:
<system_message>, <problematic>

Markup errors are handled according to the specification in PEP 258.