rinohtype

User Manual

Release 0.5.3

Contents

Introduction 1

rinohtype was initially conceived as a modern replacement for LaTeX. An important goal in the design of rinohtype is for documents to be much easier to customize than in LaTeX. By today's standards, the arcane TeX macro language upon which LaTeX is built makes customization unnecessarily difficult for one. Simply being built with Python makes rinohtype already much easier to approach than TeX. Additionally, rinohtype is built around the following core concepts to ensure customizability:

Document Templates
: These determine the page layout and (for longer documents) the different parts of your document. The templates included with rinohtype are highly configurable and allow changing margins, headers, footers, chapter titles, etc. If this is not sufficient, a custom template can be created.

Style Sheets
: The CSS-inspired style sheets determine the look of individual document elements. A style sheet assigns style attributes to each type of document element. For example, a paragraph's style is determined by the typeface, font weight, size and color, horizontal alignment of text etc.

Structured Input
: rinohtype renders a document from a document tree that does not describe any style aspects but only semantics. The style sheet maps specific style properties to the elements in this document tree. The document tree can be automatically generated from a structured document format such as reStructuredText and CommonMark using one of the included frontends, or it can be constructed manually.

rinohtype is implemented as a Python package and doubles as a high-level PDF library. Its modular design makes it easy to to customize and extend for specific applications. Moreover, because rinotype's source code is open, all of its internals can be inspected and even modified, making it customizable at all levels.

1.1 Usage Examples

rinohtype supports three modes of operation, which are discussed in more detail in the Chapter 3 guide. For each of these modes, you can choose to use one of the document templates included with rinohtype or a third-party template available from PyPI and optionally customize it to your needs. Or you can create a custom template from scratch. The same is true for the style sheet used to style the document elements.

1.1.1 Command-Line Renderer

rinohtype includes the **`rinoh`** command-line tool which renders structured text documents. Currently, reStructuredText and CommonMark documents are supported in the open-source

version. Support for DITA is available in the commercially supported Pro version.

Rendering the reStructuredText demonstration article demo.txt (using the standard article template and style sheet) generates `demo.pdf`.

1.1.2 Sphinx Builder

Configuring rinohtype as a builder for Sphinx allows rendering a Sphinx project to PDF without the need for a LaTeX installation. The document you are reading was rendered using rinohtype's Sphinx builder.

1.1.3 High-level PDF library

rinohtype can also be used as a Python library to generate PDF documents. Just like with **`rinoh`** and the Sphinx builder, you can select which document template and style sheet to use.

Additionally, you need to supply a document tree. This document tree can be parsed from a structured document format such as reStructuredText by using one of the provided frontends or built manually using building blocks provided by rinohtype. You can also write a frontend for a custom format such as an XML dialect.

All of these approaches allow for parts of the content to be fetched from a database or other data sources. When parsing the document tree from a structured document format, a templating engine like Jinja2 can be used.

Todo

sample documents

Installation 2

rinohtype supports Python 3.3 and up. Use pip to install the latest version of rinohtype and its dependencies:

```
pip install rinohtype
```

If you plan on using rinohtype as an alternative to LaTeX, you will want to install Sphinx as well:

```
pip install Sphinx
```

See Section 3.2 in the Chapter 3 guide on how to render Sphinx documents with rinohtype.

2.1 Dependencies

For parsing reStructuredText and CommonMark documents, rinohtype depends on docutils and recommonmark respectively. pip takes care of these requirements automatically when you install rinohtype.

If you want to include images other than PDF, PNG or JPEG, you need to install Pillow additionally.

Quickstart 3

This section gets you started quickly, discussing each of the three modes of operation introduced in Chapter 1. If you want to customize the style of the PDF document, please refer to Chapter 4 which introduces style sheets and document templates.

3.1 Command-Line Renderer

Installing rinohtype places the **rinoh** script in the `PATH`. This can be used to render structured documents such as demo.txt (reStructuredText):

```
rinoh --format reStructuredText demo.txt
```

After rendering finishes, you will find `demo.pdf` alongside the input file.

rinoh allows specifying the document template and style sheet to use when rendering the reStructuredText document. See its command-line options for details.

Two rendering passes are required to make sure that cross-references to page numbers are correct. After a document has been rendered, rinohtype will save the page reference data to a *.rtc* file. Provided the document (or the template or style sheet) doesn't change a lot, this can prevent the need to perform a second rendering pass.

3.2 Sphinx Builder

If your Sphinx project is already configured for the LaTeX builder, rinohtype will happily interpret `latex_documents`. Otherwise, you need to set the `rinoh_documents` configuration option:

```
rinoh_documents = [dict(doc='index',        # top-level file (index.rst)
                        target='manual')]   # output file (manual.pdf)
```

The dictionary accepts optional keys in addition to the required *doc* and *target* keys. See sphinx_builder for details.

When building the documentation, select the *rinoh* builder by passing it to the `sphinx-build -b` option:

```
sphinx-build -b rinoh . _build/rinoh
```

Note that, just like the **rinoh** command line tool, the Sphinx builder requires two rendering passes.

3.3 High-level PDF Library

Note

The focus of rinohtype development is currently on the **`rinoh`** tool and Sphinx builder. Use as a Python library is possible, but documentation may be lacking. Please be patient.

The most basic way to use rinohtype in an application is to hook up an included frontend, a document template and a style sheet:

```
from rinoh.frontend.rst import ReStructuredTextReader
from rinoh.templates import Article

# the parser builds a rinohtype document tree
parser = ReStructuredTextReader()
with open('my_document.rst') as file:
    document_tree = parser.parse(file)

# render the document to 'my_document.pdf'
document = Article(document_tree)
document.render('my_document')
```

This basic application can be customized to your specific requirements by customizing the document template, the style sheet and the way the document's content tree is built. The basics of document templates and style sheets are covered in later sections.

The document tree returned by the `ReStructuredTextReader` in the example above can also be built manually. A `DocumentTree` is simply a list of `Flowables`, which can have child elements. These children in turn can also have children, and so on; together they form a tree.

Here is an example document tree of a short article:

```
from rinoh.document import DocumentTree
from rinoh.styleds import *

document_tree = DocumentTree(
                    [Paragraph('My Document', style='title'), # metadata!
                     Section([Heading('First Section'),
                              Paragraph('This is a paragraph with some '
                                        + StyledText('emphasized text',
                                                     style='emphasis')
                                        + ' and an '
                                        + InlineImage('image.pdf')),
                              Section([Heading('A subsection'),
                                       Paragraph('Another paragraph')
                                      ])
                             ]),
                     Section([Heading('Second Section'),
                              List([Paragraph('a list item'),
                                    Paragraph('another list item')
                                   ])
                             ])
                    ])
```

It is clear that this type of content is best parsed from a structured document format such as reStructuredText or XML. Manually building a document tree is well suited for short, custom

documents however.

Basic Document Styling 4

rinohtype allows for fine-grained control over the style of its output. Most aspects of a document's style can be controlled by style sheet files and template configuration files which are being introduced in this chapter. These files are plain text files that are easy to create, read and modify.

4.1 Style Sheets

A style sheet defines the look of each element in a document. For each type of document element, the style sheet assign values to the style properties available for that element. Style sheets are stored in plain text files using the Windows INI[1] format with the `.rts` extension. Below is an excerpt from the sphinx_stylesheet included with rinohtype.

```
[STYLESHEET]
name=Sphinx
description=Mostly a copy of the LaTeX style included with Sphinx
pygments_style=friendly

[VARIABLES]
mono_typeface=TeX Gyre Cursor
serif_typeface=TeX Gyre Pagella
sans_typeface=Tex Gyre Heros
fallback_typeface=DejaVu Serif
thin_black_stroke=0.5pt,#000
blue=#20435c

[default:Paragraph]
typeface=$(serif_typeface)
font_weight=REGULAR
font_size=10pt
line_spacing=fixed(12pt, leading(0))
indent_first=0
space_above=0
space_below=0
text_align=JUSTIFY
kerning=True
ligatures=True
hyphen_lang=en_US
hyphen_chars=4

[fallback]
typeface=$(fallback_typeface)
```

1 see *Supported INI File Structure* in `configparser`

```
[body]
base=default
space_above=5pt
space_below=0
text_align=justify

[emphasis]
font_slant=italic

[strong]
font_weight=BOLD

[literal emphasis]
base=emphasis
typeface=$(mono_typeface)
hyphenate=False
ligatures=False

[literal strong]
base=strong
typeface=$(mono_typeface)
hyphenate=False
ligatures=False

[inline math]
base=monospaced

[quote]
font_slant=italic
```

Except for `[STYLESHEET]` and `[VARIABLES]`, each configuration section in a style sheet determines the style of a particular type of document element. The `emphasis` style, for example, determines the look of emphasized text, which is displayed in an italic font. This is similar to how HTML's cascading style sheets work. In rinohtype however, document elements are identified by means of a descriptive label (such as *emphasis*) instead of a cryptic selector. rinohtype also makes use of selectors, but these are collected in a Section 5.3 which maps them to descriptive names to be used by many style sheets. Unless you are using rinohtype as a PDF library to create custom documents, the default matcher should cover your needs.

The following two subsections illustrate how to extend an existing style sheet and how to create a new, independent style sheet. For more in-depth information on style sheets, please refer to Chapter 5.

4.1.1 Extending an Existing Style Sheet

Starting from an existing style sheet, it is easy to make small changes to the style of individual document elements. The following style sheet file is based on the Sphinx stylesheet included with rinohtype.

```
[STYLESHEET]
name=My Style Sheet
description=Small tweaks made to the Sphinx style sheet
base=sphinx

[VARIABLES]
mono_typeface=Courier
```

```
[emphasis]
font_color=#00a

[strong]
base=DEFAULT_STYLE
font_color=#a00
```

By default, styles defined in a style sheet *extend* the corresponding style from the base style sheet. In this example, emphasized text will be set in an italic font (as configured in the base style sheet) and colored blue (`#00a`).

It is also possible to completely override a style definition. This can be done by setting the `base` of a style definition to `DEFAULT_STYLE` as illustrated by the *strong* style. This causes strongly emphasised text to be displayed in red (#a00) but **not** in a bold font as was defined in the base style sheet (the default for `font_weight` is *Medium*; see `TextStyle`). Refer to default_matcher to find out which style attributes are accepted by each style (by following the hyperlink to the style class's documentation).

The style sheet also redefines the `mono_typeface` variable. This variable is used in the base style sheet in all style definitions where a monospaced font is desired. Redefining the variable in the derived style sheet affects all of these style definitions.

4.1.2 Starting from Scratch

If you don't specify a base style sheet in the `[STYLESHEET]` section, you create an independent style sheet. You should do this if you want to create a document style that is not based on an existing style sheet. If the style definition for a particular document element is not included in the style sheet, the default values for its style properties are used.

Todo

specifying a custom matcher for an INI style sheet

Unless a custom `StyledMatcher` is passed to `StyleSheetFile`, the default matcher is used. Providing your own matcher offers even more customizability, but it is unlikely you will need this. See Section 5.3.

Note

In the future, rinohtype will be able to generate an empty INI style sheet, listing all styles defined in the matcher with the supported style attributes along with the default values as comments. This generated style sheet can serve as a good starting point for developing a custom style sheet from scratch.

4.2 Document Templates

As with style sheets, you can choose to make use of a template provided by rinohtype and optionally customize it or you can create a custom template from scratch. This section discusses how you can configure an existing template. See Chapter 6 on how to create a custom template.

4.2.1 Configuring a Template

rinohtype provides a number of standard_templates. These can be customized by means of a template configuration file; a plain text file in the INI[1] format with the `.rtt` extension. Here is

an example configuration for the article template:

```
[TEMPLATE_CONFIGURATION]
name = my article configuration
template = article

parts =
    title
    ;front_matter
    contents
stylesheet = sphinx_base14
language = fr
abstract_location = title

[SectionTitles]
contents = 'Contents'

[AdmonitionTitles]
caution = 'Careful!'
warning = 'Please be warned'

[VARIABLES]
paper_size = A5

[title]
page_number_format = lowercase roman
end_at_page = left

[contents]
page_number_format = number

[title_page]
top_margin = 2cm
```

The `TEMPLATE_CONFIGURATION` sections collects global template options. Set *name* to provide a short label for your template configuration. *template* identifies the document template to configure.

All document templates consist of a number of document parts. The `Article` template defines three parts: `title`, `front_matter` and `contents`. The order of these parts can be changed (although that makes little sense for the article template), and individual parts can optionally be hidden by setting the `parts` configuration option. The configuration above hides the front matter part (commented out using a semicolon), for example.

The template configuration also specifies which style sheet is used for styling document elements. The `DocumentTemplate.stylesheet` option takes the name of an installed style sheet (see `rinoh --list-stylesheets`) or the filename of a stylesheet file (`.rts`).

The `language` option sets the default language for the document. It determines which language is used for standard document strings such as section and admonition titles.

The `Article` template defines two custom template options. The `abstract_location` option determines where the (optional) article abstract is placed, on the title page or in the front matter part. `table_of_contents` allows hiding the table of contents section. Empty document parts will not be included in the document. When the table of contents section is suppressed and there is no abstract in the input document or `abstract_location` is set to title, the front matter document part will not appear in the PDF.

The standard document strings configured by the `language` option described above can be overridden by user-defined strings in the `SectionTitles` and `AdmonitionTitles` sections of the configuration file. For example, the default title for the table of contents section (*Table of*

Contents) is replaced with *Contents*. The configuration also sets custom titles for the caution and warning admonitions.

The others sections in the configuration file are the `VARIABLES` section, followed by document part and page template sections. Similar to style sheets, the variables can be referenced in the template configuration sections. Here, the `paper_size` variable is set, which is being referenced by by all page templates in `Article` (although indirectly through the `page` base page template).

For document part templates, `page_number_format` determines how page numbers are formatted. When a document part uses the same page number format as the preceding part, the numbering is continued.

The `DocumentPartTemplate.end_at_page` option controls at which page the document part ends. This is set to `left` for the title part in the example configuration to make the contents part start on a right page.

Each document part finds page templates by name. They will first look for specific left/right page templates by appending `_left_page` or `_right_page` to the document part name. If these page templates have not been defined in the template, it will look for the more general `<document part name>_page` template. Note that, if left and right page templates have been defined by the template (such as the book template), the configuration will need to override these, as they will have priority over the general page template defined in the configuration.

The example configuration only adjusts the top margin for the `TitlePageTemplate`, but many more aspects of the page templates are configurable. Refer to standard_templates for details.

Todo

base for part template?

4.2.2 Using a Template Configuration File

A template configuration file can be specified when rendering using the command-line **`rinoh`** tool by passing it to the `--template` command-line option. When using the Sphinx_builder, you can set the `rinoh_template` option in `conf.py`.

To render a document using this template configuration programatically, load the template file using `TemplateConfigurationFile`:

```
import sys
from pathlib import Path

from rinoh.frontend.rst import ReStructuredTextReader
from rinoh.template import TemplateConfigurationFile

# the parser builds a rinohtype document tree
parser = ReStructuredTextReader()
with open('my_document.rst') as file:
    document_tree = parser.parse(file)

# load the article template configuration file
script_path = Path(sys.path[0]).resolve()
config = TemplateConfigurationFile(script_path / 'my_article.rtt')

# render the document to 'my_document.pdf'
document = config.document(document_tree)
document.render('my_document')
```

The `TemplateConfigurationFile.document()` method creates a document instance with the template configuration applied. So if you want to render your document using a different template configuration, it suffices to load the new configuration file.

Refer to the `Article` documentation to discover all of the options accepted by it and the document part and page templates.

Element Styling 5

This section describes how styles defined in a style sheet are applied to document elements. Understanding how this works will help you when designing a custom style sheet.

rinohtype's style sheets are heavily inspired by CSS, but add some additional functionality. Similar to CSS, rinohtype makes use of so-called *selectors* to select document elements in the *document tree* to style. Unlike CSS however, these selectors are not directly specified in a style sheet. Instead, all selectors are collected in a *matcher* where they are mapped to descriptive labels for the selected elements. A *style sheet* assigns style properties to these labels. Besides the usefulness of having these labels instead of the more cryptic selectors, a matcher can be reused by multiple style sheets, avoiding duplication.

Note

This section currently assumes some Python or general object-oriented programming knowledge. A future update will move Python-specific details to another section, making things more accessible for non-programmers.

5.1 Document Tree

A `Flowable` is a document element that is placed on a page. It is usually a part of a document tree. Flowables at one level in a document tree are rendered one below the other.

Here is schematic representation of an example document tree:

```
|- Section
|   |- Paragraph
|   \- Paragraph
\- Section
    |- Paragraph
    |- List
    |   |- ListItem
    |   |   |- Paragraph (item label; a number or bullet symbol)
    |   |   \- StaticGroupedFlowables (item body)
    |   |       \- Paragraph
    |   \- ListItem
    |       \- Paragraph
    |   |   \- StaticGroupedFlowables
    |   |       \- List
    |   |           |- ListItem
    |   |           |   \- ...
    |   |           \- ...
    \- Paragraph
```

This represents a document consisting of two sections. The first section contains two paragraphs. The second section contains a paragraph followed by a list and another paragraph. All of the elements in this tree are instances of `Flowable` subclasses.

`Section` and `List` are subclasses of `GroupedFlowables`; they group a number of flowables. In the case of `List`, these are always of the `ListItem` type. Each list item contains an item number (ordered list) or a bullet symbol (unordered list) and an item body. For simple lists, the item body is typically a single `Paragraph`. The second list item contains a nested `List`.

A `Paragraph` does not have any `Flowable` children. It is however the root node of a tree of *inline elements*. This is an example paragraph in which several text styles are combined:

```
Paragraph
 |- SingleStyledText('Text with ')
 |- MixedStyledText(style='emphasis')
 |    |- SingleStyledText('multiple ')
 |    \- MixedStyledText(style='strong')
 |         |- SingleStyledText('nested ')
 |         \- SingleStyledText('styles', style='small caps')
 \- SingleStyledText('.')
```

The visual representation of the words in this paragraph is determined by the applied style sheet. Read more about how this works in the next section.

Besides `SingleStyledText` and `MixedStyledText` elements (subclasses of `StyledText`), paragraphs can also contain `InlineFlowable`s. Currently, the only inline flowable is `InlineImage`.

The common superclass for flowable and inline elements is `Styled`, which indicates that these elements can be styled using the style sheets.

5.2 Selectors

Selectors in rinohtype select elements of a particular type. The *class* of a document element serves as a selector for all instances of the class (and its subclasses). The `Paragraph` class is a selector that matches all paragraphs in the document, for example:

```
Paragraph
```

As with CSS selectors, elements can also be matched based on their context. For example, the following matches any paragraph that is a direct child of a list item or in other words, a list item label:

```
ListItem / Paragraph
```

Python's ellipsis can be used to match any number of levels of elements in the document tree. The following selector matches paragraphs at any level inside a table cell:

```
TableCell / ... / Paragraph
```

To help avoid duplicating selector definitions, context selectors can reference other selectors defined in the same Section 5.3 using `SelectorByName`:

```
SelectorByName('definition term') / ... / Paragraph
```

Selectors can select all instances of `Styled` subclasses. These include `Flowable` and `StyledText`, but also `TableSection`, `TableRow`, `Line` and `Shape`. Elements of some of the latter classes only appear as children of other flowables (such as `Table`).

Similar to a HTML element's *class* attribute, `Styled` elements can have an optional *style*

attribute which can be used when constructing a selector. This one selects all styled text elements with the *emphasis* style, for example:

```
StyledText.like('emphasis')
```

The `Styled.like()` method can also match **arbitrary attributes** of elements by passing them as keyword arguments. This can be used to do more advanced things such as selecting the background objects on all odd rows of a table, limited to the cells not spanning multiple rows:

```
TableCell.like(row_index=slice(0, None, 2), rowspan=1) / TableCellBackground
```

The argument passed as *row_index* is a slice object that is used for extended indexing[2]. To make this work, `TableCell.row_index` is an object with a custom `__eq__()` that allows comparison to a slice.

Rinohtype borrows CSS's concept of specificity to determine the "winning" selector when multiple selectors match a given document element. Each part of a selector adds to the specificity of a selector. Roughly stated, the more specific selector will win. For example:

```
ListItem / Paragraph                        # specificity (0, 0, 0, 0, 2)
```

wins over:

```
Paragraph                                   # specificity (0, 0, 0, 0, 1)
```

since it matches two elements instead of just one.

Specificity is represented as a 5-tuple. The last four elements represent the number of *location* (currently not used), *style*, *attribute* and *class* matches. Here are some selectors along with their specificity:

```
StyledText.like('emphasis')                 # specificity (0, 0, 1, 0, 1)
TableCell / ... / Paragraph                 # specificity (0, 0, 0, 0, 2)
TableCell.like(row_index=2, rowspan=1)      # specificity (0, 0, 0, 2, 1)
```

Specificity ordering is the same as tuple ordering, so (0, 0, 1, 0, 0) wins over (0, 0, 0, 5, 0) and (0, 0, 0, 0, 3) for example. Only when the number of style matches are equal, the attributes match count is compared and so on.

In practice, the class match count is dependent on the element being matched. If the class of the element exactly matches the selector, the right-most specificity value is increased by 2. If the element's class is a subclass of the selector, it is only increased by 1.

The first element of the specificity tuple is the *priority* of the selector. For most selectors, the priority will have the default value of 0. The priority of a selector only needs to be set in some cases. For example, we want the `CodeBlock` selector to match a `CodeBlock` instance. However, because `CodeBlock` is a `Paragraph` subclass, another selector with a higher specificity will also match it:

```
CodeBlock                                   # specificity (0, 0, 0, 0, 2)
DefinitionList / Definition / Paragraph     # specificity (0, 0, 0, 0, 3)
```

To make sure the `CodeBlock` selector wins, we increase the priority of the `CodeBlock` selector by prepending it with a + sign:

```
+CodeBlock                                  # specificity (1, 0, 0, 0, 2)
```

In general, you can use multiple + or − signs to adjust the priority:

```
++CodeBlock                                 # specificity (2, 0, 0, 0, 2)
---CodeBlock                                # specificity (-3, 0, 0, 0, 2)
```

2 Indexing a list like this `lst[slice(0, None, 2)]` is equivalent to `lst[0::2]`.

5.3 Matchers

At the most basic level, a `StyledMatcher` is a dictionary that maps labels to selectors:

```
matcher = StyledMatcher()
...
matcher['emphasis'] = StyledText.like('emphasis')
matcher['chapter'] = Section.like(level=1)
matcher['list item number'] = ListItem / Paragraph
matcher['nested line block'] = (GroupedFlowables.like('line block')
                                / GroupedFlowables.like('line block'))
...
```

Rinohtype currently includes one matcher which defines labels for all common elements in documents:

```
from rinoh.stylesheets import matcher
```

5.4 Style Sheets

A `StyleSheet` takes a `StyledMatcher` to provide element labels to assign style properties to:

```
styles = StyleSheet('IEEE', matcher=matcher)
...
styles['strong'] = TextStyle(font_weight=BOLD)
styles('emphasis', font_slant=ITALIC)
styles('nested line block', margin_left=0.5*CM)
...
```

Each `Styled` has a `Style` class associated with it. For `Paragraph`, this is `ParagraphStyle`. These style classes determine which style attributes are accepted for the styled element. Because the style class can automatically be determined from the selector, it is possible to simply pass the style properties to the style sheet by calling the `StyleSheet` instance as shown above.

Style sheets are usually loaded from a *.rts* file using `StyleSheetFile`. An example style sheet file is shown in Section 4.1.

A style sheet file contains a number of sections, denoted by a section title enclosed in square brackets. There are two special sections:

- `[STYLESHEET]` describes global style sheet information (see `StyleSheetFile` for details)
- `[VARIABLES]` collects variables that can be referenced elsewhere in the style sheet

Other sections define the style for a document elements. The section titles correspond to the labels associated with selectors in the `StyledMatcher`. Each entry in a section sets a value for a style attribute. The style for enumerated lists is defined like this, for example:

```
[enumerated list]
margin_left=8pt
space_above=5pt
space_below=5pt
ordered=true
flowable_spacing=5pt
number_format=NUMBER
label_suffix=')'
```

Since this is an enumerated list, *ordered* is set to `true`. *number_format* and *label_suffix* are set to produce list items labels of the style *1), 2),* …. Other entries control margins and spacing. See `ListStyle` for the full list of accepted style attributes.

Todo

base stylesheets are specified by name … entry points

5.4.1 Base Styles

It is possible to define styles which are not linked to a selector. These can be useful to collect common attributes in a base style for a set of style definitions. For example, the Sphinx style sheet defines the *header_footer* style to serve as a base for the *header* and *footer* styles:

```
[header_footer : Paragraph]
base=default
typeface=$(sans_typeface)
font_size=10pt
font_weight=BOLD
indent_first=0pt
tab_stops=50% CENTER, 100% RIGHT

[header]
base=header_footer
padding_bottom=2pt
border_bottom=$(thin_black_stroke)
space_below=24pt

[footer]
base=header_footer
padding_top=4pt
border_top=$(thin_black_stroke)
space_above=18pt
```

Because there is no selector associated with *header_footer*, the element type needs to be specified manually. This is done by adding the name of the relevant `Styled` subclass to the section name, using a colon (`:`) to separate it from the style name, optionally surrounded by spaces.

5.4.2 Custom Selectors

It is also possible to define new selectors directly in a style sheet file. This allows making tweaks to an existing style sheet without having to create a new `StyledMatcher`. However, this should be used sparingly. If a great number of custom selectors are required, it is better to create a new `StyledMatcher`

The syntax for specifying a selector for a style is similar to that when constructing selectors in a Python source code (see Section 5.3), but with a number of important differences. A `Styled` subclass name followed by parentheses represents a simple class selector (without context). Arguments to be passed to `Styled.like()` can be included within the parentheses.

```
[special text : StyledText('special')]
font_color=#FF00FF

[accept button : InlineImage(filename='images/ok_button.png')]
baseline=20%
```

Even if no arguments are passed to the class selector, it is important that the class name is followed by parentheses. If the parentheses are omitted, the selector is not registered with the matcher and the style can only be used as a base style for other style definitions (see Section 5.4.1).

As in Python source code, context selectors are constructed using forward slashes (/) and the ellipsis (...). Another selector can be referenced in a context selector by enclosing its name in single or double quotes.

```
[admonition title colon : Admonition / ... / StyledText('colon')]
font_size=10pt

[chapter title : LabeledFlowable('chapter title')]
label_spacing=1cm
align_baselines=false

[chapter title number : 'chapter title' / Paragraph('number')]
font_size=96pt
text_align=right
```

5.4.3 Variables

Variables can be used for values that are used in multiple style definitions. This example declares a number of typefaces to allow easily replacing the fonts in a style sheet:

```
[VARIABLES]
mono_typeface=TeX Gyre Cursor
serif_typeface=TeX Gyre Pagella
sans_typeface=Tex Gyre Heros
thin_black_stroke=0.5pt,#000
blue=#20435c
```

It also defines the *thin_black_stroke* line style for use in table and frame styles, and a specific color labelled *blue*. These variables can be referenced in style definitions as follows:

```
[code block]
typeface=$(mono_typeface)
font_size=9pt
text_align=LEFT
indent_first=0
space_above=6pt
space_below=4pt
border=$(thin_black_stroke)
padding_left=5pt
padding_top=1pt
padding_bottom=3pt
```

Another stylesheet can inherit (see below) from this one and easily replace fonts in the document by overriding the variables.

5.4.4 Style Attribute Resolution

The element styling system makes a distinction between text (inline) elements and flowables with respect to how attribute values are resolved.

Text elements by default inherit the properties from their parent. Take for example the *emphasis* style definition from the example above. The value for style properties other than *font_slant*

(which is defined in the *emphasis* style itself) will be looked up in the style definition corresponding to the parent element, which can be either another `StyledText` instance, or a `Paragraph`. If the parent element is a `StyledText` that neither defines the style attribute, lookup proceeds recursively, moving up in the document tree.

For **flowables**, there is no fall-back to the parent element's style by default.

Style definitions for both types of elements accept a *base* attribute. When set, attribute lookup is first attempted in the referenced style before fallback to the parent element's style or default style (described above). This can help avoid duplication of style information and the resulting maintenance difficulties. In the following example, the *unnumbered heading level 1* style inherits all properties from *heading level 1*, overriding only the *number_format* attribute:

```
[heading level 1]
typeface=$(sans_typeface)
font_weight=BOLD
font_size=16pt
font_color=$(blue)
line_spacing=SINGLE
space_above=18pt
space_below=12pt
number_format=NUMBER
label_suffix=' '

[unnumbered heading level 1]
base=heading level 1
number_format=None
```

In addition to the names of other styles, the *base* attribute accepts two special values:

`NEXT_STYLE`
: If a style attribute is not set for the style, lookup will continue in the next style matching the element (with a lower specificity). This is similar to how CSS works. You can use this together with the + priority modifier to override some attributes for a set of elements that match different styles.

`PARENT_STYLE`
: This enables fallback to the parent element's style for flowables. Note that this requires that the current element type is the same or a subclass of the parent type, so it cannot be used for all styles.

When a value for a particular style attribute is set nowhere in the style definition lookup hierarchy, its default value is returned. The default values for all style properties are defined in the class definition for each of the `Style` subclasses.

5.5 Style Logs

When rendering a document, rinohtype will create a style log. It is written to disk using the same base name as the output file, but with a *.stylelog* extension. The information logged in the style log is invaluable when developing a style sheet. It tells you which style maps to each element in the document.

The style log lists the document elements that have been rendered to each page as a tree. The corresponding location of each document element in the source document is displayed on the same line, if the frontend provides this information. This typically includes the filename, line number and possibly other information such as a node name.

For each document element, all matching styles are listed together with their specificity, ordered from high to low specificity. No styles are listed when there aren't any selectors matching the element; the default attribute value is used (for flowables) or looked up in the parent element(s) (for text/inline elements). See Section 5.4.4 for specifics.

The winning style is indicated with a `>` symbol in front of it. Styles that are not defined in the style sheet or its base(s) are marked with an `x`. Each style name is followed by the (file)name of the top-most stylesheet where it is defined (within brackets) and the name of its base style (after `>`).

Here is an example excerpt from a style log:

```
...
  Paragraph('January 03, 2012', style='title page date')
       > (0,0,1,0,2) title page date [Sphinx] > DEFAULT
         (0,0,0,0,2) body [Sphinx] > default
    SingleStyledText('January 03, 2012')
--------------------------------- page 3 ---------------------------------
#### FootnoteContainer('footnotes')
  StaticGroupedFlowables()
#### DownExpandingContainer('floats')
  StaticGroupedFlowables()
#### ChainedContainer('column1')
  DocumentTree(id='restructuredtext-demonstration')
    Section(id='structural-elements')           demo.txt:62 <section>
         > (0,0,0,1,4) content chapter [Sphinx] > chapter
           (0,0,0,1,2) chapter [Sphinx] > DEFAULT
      Heading('1 Structural Elements')          demo.txt:62 <title>
           > (0,0,0,1,2) heading level 1 [Sphinx] > DEFAULT
             (0,0,0,0,2) other heading levels [Sphinx] > heading level 5
        MixedStyledText('1 ')
          SingleStyledText('1')
          SingleStyledText(' ')
        SingleStyledText('Structural Elements')
      Paragraph('A paragraph.')                 demo.txt:64 <paragraph>
           > (0,0,0,0,2) body
        MixedStyledText('A paragraph.')
          SingleStyledText('A paragraph.')
      List(style='bulleted')                    demo.txt:124 <bullet_list>
           > (0,0,1,0,2) bulleted list [Sphinx] > enumerated list
        ListItem()                              None:None <list_item>
             x (0,0,1,0,4) bulleted list item
          ListItemLabel('•')
               > (0,0,1,0,6) bulleted list item label [Sphinx] > list item
label
                 (0,0,0,0,2) list item label [Sphinx] > default
            SingleStyledText('•')
          StaticGroupedFlowables()
               > (0,0,0,0,3) list item body [Sphinx] > DEFAULT
            Paragraph('A bullet list')          demo.txt:124 <paragraph>
                 > (0,0,0,0,5) list item paragraph [Sphinx] > default
                   (0,0,0,0,2) body [Sphinx] > default
              MixedStyledText('A bullet list')
                MixedStyledText('A bullet list')
                  SingleStyledText('A bullet list')
            List(style='bulleted')              demo.txt:126 <bullet_list>
                 > (0,0,1,0,5) nested bulleted list [Sphinx] > bulleted list
                   (0,0,1,0,2) bulleted list [Sphinx] > enumerated list
              ListItem()                        None:None <list_item>
                   x (0,0,1,0,4) bulleted list item
                ListItemLabel('•')
                     > (0,0,1,0,6) bulleted list item label [Sphinx] > list
item label
```

```
                        (0,0,0,0,2) list item label [Sphinx] > default
                  SingleStyledText('•')
                StaticGroupedFlowables()
                     > (0,0,0,0,3) list item body [Sphinx] > DEFAULT
...
```

Document Templates 6

When it is not possible to achieve a particular document style using one of the existing templates and a custom template configuration, you can create a new template. A new template is programmed in Python and therefor it is required that you are familiar with Python, or at least with general object-oriented programming.

6.1 Subclassing a Template

If you need to customize a template beyond what is possible by configuration, you can subclass a template class and override document part and page templates with custom templates. The following example subclasses `Article`.

```
from rinoh.attribute import OverrideDefault
from rinoh.template import DocumentPartTemplate, PageTemplate
from rinoh.templates import Article


class BibliographyPartTemplate(DocumentPartTemplate):
    ...


class MyTitlePageTemplate(PageTemplate):
    ...


class MyArticle(Article):
    parts = OverrideDefault(['title', 'contents', 'bibliography'])

    # default document part templates
    bibliography = BibliographyPartTemplate()

    # default page templates
    title_page = MyTitlePageTemplate(base='page')
    bibliography_page = PageTemplate(base='page')
```

`MyArticle` extends the `Article` template, adding the extra `bibliography` document part, along with the page template `bibliography_page`. The new document part is included in `parts`, while also leaving out `front_matter` by default. Finally, the template also replaces the title page template with a custom one.

6.2 Creating a Custom Template

A new template can be created from scratch by subclassing `DocumentTemplate`, defining all document parts, their templates and page templates.

The `Article` and `Book` templates are examples of templates that inherit directly from `DocumentTemplate`. We will briefly discuss the article template. The `Article` template overrides the default style sheet and defines the two custom template attributes discussed in Section 4.2.1. The document parts `title`, `front_matter` and `contents` are listed the in `parts` attribute and part templates for each are provided along with page templates:

```
class Article(DocumentTemplate):
    stylesheet = OverrideDefault(sphinx_article)
    table_of_contents = Attribute(Bool, True,
                                  'Show or hide the table of contents')
    abstract_location = Attribute(AbstractLocation, 'front matter',
                                  'Where to place the abstract')

    parts = OverrideDefault(['title', 'front_matter', 'contents'])

    # default document part templates
    title = TitlePartTemplate()
    front_matter = ArticleFrontMatter(page_number_format='continue')
    contents = ContentsPartTemplate(page_number_format='continue')

    # default page templates
    page = PageTemplate(page_size=Var('paper_size'))
    title_page = TitlePageTemplate(base='page',
                                   top_margin=8*CM)
    front_matter_page = PageTemplate(base='page')
    contents_page = PageTemplate(base='page')
```

The custom `ArticleFrontMatter` template reads the values for the two custom template attributes defined in `Article` to determine which flowables are included in the front matter:

```
class ArticleFrontMatter(DocumentPartTemplate):
    toc_section = TableOfContentsSection()

    def _flowables(self, document):
        meta = document.metadata
        abstract_loc = document.get_option('abstract_location')
        if ('abstract' in meta
                and abstract_loc == AbstractLocation.FRONT_MATTER):
            yield meta['abstract']
        if document.get_option('table_of_contents'):
            yield self.toc_section
```

Have a look at the src/book for an example of a slightly more complex template that defines separate templates for left and right pages.

Frequently Asked Questions 7

Below is the start of a list of commonly encountered problems and solutions to them. You can also find answers to usage questions on rinohtype on StackOverflow.

PDFs produced by rinohtype contain mostly empty pages. What's up?

Old versions of some PDF viewers do not support the way rinohtype embeds fonts in a PDF (see issue 2). PDF viewers that are known to be affected are:

- pre-37.0 Firefox's built-in PDF viewer (pdf.js)
- pre-0.41 poppler-based applications such as Evince

Contributing 8

Thank you for considering contributing to rinohtype! This document contains some general information with respect to contributions. Please refer to DEVELOPING.rst for specific information on running tests etc.

8.1 Contributor License Agreement

Please send your contributions by creating a pull request on GitHub. You will be asked to agree to the contributor license agreement. Why, you might ask? In the past, I had plans to sell a commercial version of rinohtype. This version would basically be the same as the open source version, but with the addition of a closed-source frontend for the DITA format. While I'm no longer actively pursuing these plans at this point, I would like to keep this option open. In combination with the Affero GPL, the CLA allows me to include contributions by others in this commercial version while preventing others from releasing commercial closed-source versions of rinohtype. Additionally, the CLA protects the project from legal risks. Since I'm no legal expert, I adapted the contributor license agreement from Project Harmony. You can find a quick summary and a detailed walkthrough of the license in the guide.

8.2 Coding guidelines

Please follow the coding style used in the existing codebase, which generally follows the PEP 8 style guide. Here are the most important rules the codebase conforms to:

- lines are wrapped at 80 columns
- 4 spaces indentation (no tabs)
- descriptive variable/function/class names (not shortened)
- imports are grouped into sections: standard library, external packages and rinohtype modules
- minimize use of external dependencies

Developing 9

This project makes use of Nox to manage running tests and other tasks. The tests can be divided into three categories: checks, unit tests and regression (integration) tests. `noxfile.py` defines "sessions" that configure these tasks. Each of these are described in the next section. The unit and regression tests make use of pytest.

The repository includes a Poetry `pyproject.toml` file to help you set up a virtual environment with Nox and other development dependencies. From the repository checkout root directory execute:

```
poetry install
```

Poetry will create a virtual environment in *.venv*, which you can activate like this (on Linux/macOS):

```
source .venv/bin/activate
```

Now Nox is available for running the tests. Alternatively, you can run Nox through Poetry:

```
poetry run nox
```

Starting `nox` without any arguments will run the *check*, *check_docs*, *unit* and *regression* sessions. Refer to the next section for an overview of all sessions.

9.1 Nox Sessions

The following sessions execute unit tests and regression tests:

`unit`
: Runs the unit tests.

`regression`
: Runs integration/regression tests. Each of these tests render a tiny document (often a single page) focusing on a specific feature, which means they also execute quickly. Their PDF output is compared to a reference PDF that is known to be good. This requires ImageMagick and either MuPDF's *mutool* or poppler's *pdftoppm* to be available from the search path.

These sessions are parametrized in order to run the tests against both the source and wheel distributions, and against all supported Python versions. Executing e.g. `nox --session unit` will run all of these combinations. Run `nox --list` to display these. You can run a single session like this: `nox --session "<session name>"`.

`unit_sphinx`, `regression_docutils`, `regression_sphinx`
: These are variations on the *unit* and *regression* sessions that run the (relevant) tests against several versions of the principal rinohtype dependencies, docutils and Sphinx (respecting

version constraints specified in `pyproject.toml`).

Note that for development purposes, it generally suffices to run the default set of sessions. Section 9.3 will run all session to catch regressions.

The other environments run checks, build documentation and build "binary" distributions for Mac and Windows:

`check`
: Performs basic checks; just `poetry check` at this point.

`check_docs`
: Perform checks on the documentation source files using doc8 and sphinx-doctest. restview can be useful when fixing syntax errors in `README.rst`, `CHANGES.rst`, …

`build_docs`
: Build the rinohtype documentation using Sphinx, both in HTML and PDF formats.

`macapp` (not maintained, broken?)
: Build a stand-alone macOS application bundle using briefcase. This task can only be run on macOS.

`wininst` (not maintained, broken?)
: Build a stand-alone rinohtype installer for the Windows platform with the help of pynsist. This task also can be run on other platforms than Windows.

Customization settings for doc8 and pytest are stored in `setup.cfg`.

9.2 Testing against multiple Python interpreter versions

Nox facilitates running tests on multiple Python interpreter versions. You can combine the `unit` and `regression` sessions with a Python version number to execute it on a specific Python interpreter version. For example, to run the unit tests with CPython 3.8:

```
nox -e unit-3.8
```

While it is typically sufficient to test on a single Python version during development, it can be useful to run tests on a set of Python versions before pushing your commits. Of course, this requires these versions to be available on your machine. It is highly recommended you use pyenv (or pyenv-win) to install and manage these. For example, to install CPython 3.8.1, you can run:

```
pyenv install 3.8.1
```

and pyenv will download, build and install this version of CPython for you. pyenv will install the different Python versions in an isolated location (typically under `~/.penv`), so they will not interfere with your system-default Python versions.

The file `.python-version` in the root of the repository specifies which Python versions pyenv should make available whenever we are inside the repository checkout directory. The file lists specific the versions of CPython rinohtype aims to support plus recent PyPy3 versions (ideally, we should closely track the latest releases). The `pyenv_install.py` script can install these for you (skipping any that are already installed).

9.3 Continuous integration

GitHub Actions automatically executes the Nox sessions when new commits are pushed to the repository. The Nox sessions are run on Linux, macOS and Windows, and run the tests against an array of Python, docutils and Sphinx versions to make sure that we don't break any corner cases. See `.github/workflows` for details.

9.4 Making a release

This is a list of steps to follow when making a new release of rinohtype. Publishing the new release to PyPI and uploading the documentation to GitHub Pages is handled by the GitHub Actions workflow.

1. Make sure your checkout is clean.
2. Run basic tests and checks locally:

   ```
   nox
   ```

3. Push your commits to master on GitHub. Don't create a tag yet!
4. Check whether all tests on GitHub Actions are green.
5. Set the release date.
   - set `__release_date__` in *src/rinoh/__init__.py* (`YYYY-MM-DD`)
   - add release date to this release's section (see other sections for examples)
6. Create a git tag: `git tag v$(poetry version --short)`
7. Push the new tag: `git push origin v$(poetry version --short)`
8. The GitHub workflow will run all Nox sessions and upload the new version to PyPI if all checks were successful.
9. Create a new release on GitHub. Include the relevant section of the changelog. Use previous releases as a template.
   - Tag version: the release's tag *vx.y.z*
   - Release title: *Release x.y.z (date)*
   - Add a link to the release on PyPI:

     ```
     Install from [PyPI](https://pypi.org/project/rinohtype/x.y.z/)
     ```

   - Copy the release notes from the change log
10. Bump version number and reset the release date to "upcoming".
    - `poetry version patch  # or 'minor'`
    - add new section at the top of the changelog
    - set `__release_date__` in *src/rinoh/__init__.py* to `'upcoming'`

Release History 10

10.1 Release 0.5.3 (in development)

New Features:

- Document part templates now accept a *page_number_prefix* (StyledText). For example, set `page_number_prefix = '{SECTION_NUMBER(1)}-'` to prefix the page number with the chapter number. You'll want to use this with the new page break options (see next item).
- The `page_break` style attribute now also accepts *left restart*, *right restart* and *any restart* values to restart page numbering
- The new *continue* page number format makes it more explicit when to not restart page numbering.
- Setting the *base* for a style to `NEXT_STYLE` proceeds to look up style attributes in the next matching style if they are undefined.
- If the *RINOH_NO_CACHE* environment variable is set, the references cache (.rtc file) won't be loaded nor saved. This is mostly useful for testing.

Changed:

- Smarter automatic sizing of table columns; don't needlessly pad columns whose contents don't require wrapping.

Fixed:

- Setting the *base* for a style to `PARENT_STYLE` results in a crash.
- docutils image directive: crash when encountering a width/height containing a decimal point (#251 by Karel Frajtak)
- docutils inline images don't support width, height and scale options

10.2 Release 0.5.2 (2021-02-24)

New Features:

- If the *RINOH_SINGLE_PASS* environment variable is set, rendering will be stopped after a single pass. This speeds up iteration when tweaking style sheets or templates.
- Sphinx builder: the `rinoh_targets` configuration variable allows limiting the documents to a subset of those listed in `rinoh_documents`.
- The 'number_format' style property can now also accept styled text strings which replace the auto-numbered label.
- Document elements (Styled objects) can more easily be matched based on their ID (or 'name' in docutils terms) by means of the *has_id* selector property.

Changed:

- docutils/Sphinx frontend: will default to referencing targets by number if possible, even if a custom label is explicitly set. This behaviour can be overridden in the style sheet by setting the *type* property of the *linked reference* style to 'custom' (see also issue #244).

Fixed:

- Sphinx style sheet: the object description is always rendered to the right of the signature, no matter how wide the signature is.
- Incorrect/useless warnings that popped up with release 0.5.1.

Part of the work included in this release was kindly sponsored by Joby Aviation.

10.3 Release 0.5.1 (2021-02-19)

New Features:

- Paragraphs can now be numbered. rinohtype also allows for referencing them by number, but docutils/Sphinx doesn't readily offer the means express that. A workaround for this will be included in a future release.

Fixed:

- Fix issues with metadata (title, author) stored in the PDF Info dictionary
- Fix handling of no-break spaces (they were rendered using the fallback font)
- When a caption occurs in an unnumbered chapter, an exception aborts rendering (even when `number_separator` style attribute is set to `None`)
- Handling of base template specified as string in a template configuration
- Table column widths entries now also accept fractions

Part of the work included in this release was kindly sponsored by Joby Aviation.

10.4 Release 0.5.0 (2021-02-03)

New Features:

- Google Fonts: if a specified typeface is not installed, rinohtype attempts to download the corresponding fonts from Google Fonts. Simply supply the font name as listed on https://-fonts.google.com as a value for the `typeface` style property.
- Table: in addition to fixed and relative-width columns, you can indicate columns to be automatically sized by specifying a value of 'auto' in the 'column_widths' style parameter in your style sheet.
- docutils frontend: support the `:align:` option to table directives, which will override the alignment set for the table in the style sheet.
- The starting number of enumerated lists in reStructuredText is respected.
- Table column widths can be specified in the style sheet, which take effect when these haven't been specified in the source document.
- Document elements now store where they have been defined (document tree, style sheet file or template configuration file); when you specify relative paths (e.g. for images), they are interpreted relative to the location of their source. This should make things more intuitive.
- The `page_break` style attribute is no longer reserved for sections; a page break can be forced before any flowable.
- Enumerated list items with a hidden label ('hide' style attribute) are no longer counted in the numbering.
- Templates and typefaces can be registered by name at runtime. This makes them referencable from template configuration and style sheet files. For example, custom templates/-

typefaces can be imported in a Sphinx project's *conf.py* (to be documented).
- It's now possible to add arbitrary reStructuredText content to the front/back matter or elsewhere by adding a `.. container::` with the 'out-of-line' class and a `:name:` to reference it by in the document template configuration, e.g. in the list of front matter flowables (to be documented).
- Selectors in style sheet files (.rts) now support boolean and 'None' values. For example, you can select StaticGroupedFlowables based on whether they have any children or not: e.g `TableCell(empty=true)` selects empty table cells.
- The document's title and author are now stored in the PDF metadata.
- "0" is now accepted as a valid value for Dimension-type attributes in style sheets and template configurations.

Changed:
- Rendering speed was more than doubled (caching)! (PR #197 by Alex Fargus)
- Sphinx frontend: `rinoh_documents` now takes a list of dictionaries, one for each PDF document to be built. This allows selecting e.g. the template and logo on a per-document level. Support for `rinoh_template`, `rinoh_stylesheet`, `rinoh_paper_size`, `rinoh_domain_indices` and `rinoh_logo` was removed. Fallback to `latex_documents` is retained. (PR #182, #192, #195, #208 and #216 by Alex Fargus)
- The default stylesheet ('Sphinx') now prevents captions from being separated from their image/table/code block (across pages).
- Font weights and widths are now internally represented by integer classes. In addition to integer values, string values are still accepted (mapped to classes).
- OpenTypeFont now determines the font weight, slant and width from the file. For backward compatibility, it still accepts these as arguments on instantiation but warns when they don't match the values stored in the font.

Fixed:
- Table column width determination was overhauled. Now fixed-width tables are supported and automatic-width columns should be handled better.
- The 'nested bulleted/enumerated list' selectors were broken; their corresponding styles were never applied
- Items inside a table cannot be referenced (issue #174)
- Sphinx frontend: fix handling of relative image paths in .rst files inside a directory in the Sphinx project root
- rinoh: fix –install-resources (broken since PyPI disabled XMLRPC searches)
- GroupedLabeledFlowables: respect label_min_width and fix a crash with respect to space_below handling
- Duplicate rendering of content in columns; if content was too small to fill the first column, it was rendered again in subsequent columns.
- Crash on encountering a style for which no selector is defined.

Part of the work included in this release was kindly sponsored by Joby Aviation.

10.5 Release 0.4.2 (2020-07-28)

New Features:
- before/after style attributes for StyledText (issue #158)
- docutils/Sphinx frontend: don't abort on encountering math/math_block, output the (LaTeX) math markup instead, along with printing a warning.
- docutils frontend: raw inline text (with `:format: 'rinoh'`) is parsed as styled text

Fixed:
- crash when the 'contents' topic has multiple IDs (issue #173)
- loading of the references cache (issue #170)

- some issues with space_below handling

10.6 Release 0.4.1 (2020-07-01)

New Features:

- UserStrings: arbitrary user-defined strings that can be defined in the template configuration or as a substitution definition in reStructuredText
- strings in a StringCollection can now be styled text
- Sphinx frontend: use the `today` and `today_fmt` configuration variables for the date on the title page
- Sphinx frontend: allow extensions access to the builder object (issue #155)
- rinoh: `--output` writes the output PDF to a specified location

Fixed:

- Regression in handling images that don't fit on the current page (issue #153)
- Fix crash when rendering local table of contents (issue #160)
- Sphinx frontend: support code-block/literalinclude with caption (issue #128)
- rinoh: variables set in a template configuration file are sometimes ignored (issue #164)
- Crash when using a font that contains unsupported lookups (issue #141)

10.7 Release 0.4.0 (2020-03-05)

New Features:

- automatically generated lists of figures and tables
- paragraphs now provide default tab stops (proportional to font size) for indentation
- stylesheet (.rts) and template configuration (.rtt) files now support specifying inline and background images (#107 and #108); to be documented
- it is now possible to specify selector priority (+-) in style sheets
- Sphinx frontend: the rinoh builder can be discovered by entry point (no more need to add 'rinoh.frontend.sphinx' to the list of extensions)
- rinoh: set a return code of 1 when one or more referenced images could not be found (issue #104)
- rinoh: introduce the `--install-resources` option to control the automatic installation of resources from PyPI
- German locale (contributed by Michael Kaiser)
- Polish locale (contributed by Mariusz Jamro)

Changed:

- Python 3.3 & 3.4 are no longer supported since they have reached end-of-life
- remove the dependency on purepng by embedding its png.py
- limit the width of images to the available width by default
- XML frontend: special case mixed content nodes
- fixes in the design of stylesheet/template code

Fixed:

- various regressions (PR #142 by Norman Lorrain)
- fix issues with variables defined in a base style sheet/template config
- various footnote rendering issues
- border width is also taken into account for flowables that are continued on a new page (#127)
- Sphinx: handle case when source_suffix is a list (PR #110 by Nick Barrett)
- incompatibility with Sphinx 1.6.1+ (latex_paper_size)

- docutils: crash when a footnote is defined in an admonition (issue #95)
- docutils: crash on encountering a raw text role (issue #99)
- docutils: 'decoration' node (header/footer) is not yet supported (issue #112)
- crash when a table cell contains (only) an image
- colours of PNG images with gamma (gAMA chunk) set are incorrect (#102)
- Sphinx: image paths with wildcard extension are not supported (#119)
- GroupedFlowables: space_below should only be considered at the end
- adapt to PEP 479 (Change StopIteration handling inside generators), the default in Python 3.7 (issue #133)
- fix compatibility with Python 3.6.7 and 3.7.1 (tokenizer changes)
- fix crash caused by Python 3.8's changes to int.__str__

10.8 Release 0.3.1 (2016-12-19)

New Features:

- rinoh is now also available as a stand-alone application for both Windows (installer) and macOS (app); they include an embedded CPython installation
- index terms can be StyledText now (in addition to str)
- the 'document author' metadata entry can now be displayed using a Field
- Sphinx frontend: support the 'desc_signature_line' node (new in Sphinx 1.5)
- rinoh –docs: open the online documentation in the default browser

Changed:

- more closely mimic the Sphinx LaTeX builder's title page (issue #60)
- there is no default for PageTemplate.chapter_title_flowables anymore since they are specific to the document template

Fixed:

- handle StyledText metadata (such as document title)
- Sphinx frontend: support the 'autosummary_toc' node
- DummyFlowable now sticks to the flowable following it (keep_with_next), so that (1) it does not break this behavior of Heading preceding it, and (2) IndexTargets do not get separated from the following flowable
- bug in LabeledFlowable that broke keep_with_next behavior
- the descender size of the last flowable in a GroupedFlowables with keep_with_next=True was getting lost
- GroupedFlowables should not mark the page non-empty; this caused empty pages before the first chapter if it is preceded by grouped DummyFlowables

10.9 Release 0.3.0 (2016-11-23)

New Features:

- support localization of standard document strings (en, fr, it, nl) (#53)
- localized strings can be overridden in the document template configuration
- make use of a fallback typeface when a glyph is not available (#55) (the 'fallback' style in the Sphinx stylesheet sets the fallback typeface)
- template configuration (INI) files: specify which document parts to include, configure document part and page templates, customize localized strings, …
- support specifying more complex selectors directly in a style sheet file
- (figure and table) captions support hierarchical numbering (see CaptionStyle)
- make the frontends independent of the current working directory
- reStructuredText: support the table :widths: option (upcoming docutils 0.13)

- Sphinx frontend: provide styles for Sphinx's inline markup roles
- rinoh (command line renderer):
  - support template configuration files
  - support file formats for which a frontend is installed (see –list-formats)
  - accept options to configure the frontend (see –list-options)
  - option to list the installed fonts (on the command line or in a PDF file)
- show the current page number as part of the rendering progress indicator
- Book template: support for setting a cover page
- frontends: raise a more descriptive exception when a document tree node is not mapped
- validate the default value passed to an Attribute
- preliminary support for writing a style sheet to an INI file, listing default values for non-specified attributes (#23)

Changed:

- rinoh: the output PDF is now placed in the current directory, not in the same directory as the input file
- Sphinx builder configuration: replace the `rinoh_document_template` and `rinoh_template_configuration` options with `rinoh_template`
- if no base is given for a style, style attribute lookup proceeds to look in the style of the same name in the base style sheet (#66)
- DEFAULT_STYLE can be used as a base style to prevent style attribute lookup in the style of the same name in the base style sheet
- rename FieldList to DefinitionList and use it to replace uses (docutils and Sphinx frontends) of the old DefinitionList (#54)
- the new DefinitionList (FieldList) can be styled like the old DefinitionList by setting max_label_width to None, 0 or a 0-valued Dimension
- figures are now non-floating by default (float placement needs more work)
- hide the index chapter when there are no index entries (#51)
- style sheets: use the default matcher if none is specified
- Sphinx style sheet: copy the admonition style from the Sphinx LaTeX builder
- Sphinx style sheet: keep the admonition title together with the body
- Sphinx style sheet: color linked references as in the LaTeX output (#62)
- Sphinx style sheet: disable hyphenation/ligatures for literal strong text
- no more DocumentSection; a document now consists of parts (containing pages)
- template configuration:
  - refer to document part templates by name so that they can be replaced
  - the list of document parts can be changed in the template configuration
  - document parts take the 'end_at_page' option (left, right, or any)
  - find (left/right) page templates via the document part name they belong to
  - fall back to <doc_part>_page when the right or left template is not found
  - each template configuration requires a name
- DocumentTree: make the `source_file` argument optional
- don't abort when the document section hierarchy is missing levels (#67)
- use the PDF backend by default (no need to specify it)
- store the unit with Dimension instances (better printing)
- rename the *float* module to *image*

Fixed:

- improve compatibility with Windows: Windows path names and file encoding
- crash if a StyledText is passed to HeadingStyle.number_separator
- GroupedLabeledFlowables label width could be unnecessarily wide
- fix and improve automatic table column sizing
- Figures can now be referenced using the 'reference' format ("Figure 1.2")

- HorizontallyAlignedFlowable: make more robust
- make document elements referenceable by secondary IDs
- reStructuredText: only the first classifier for a definition term was shown
- Sphinx frontend: support the 'centered' directive
- Sphinx frontend: basic support for the 'hlist' directive
- Sphinx frontend: handle :abbr: without explanation
- Sphinx frontend: support nested inline nodes (guilabel & samp roles)
- PDF backend: fix writing of Type 1 fonts from a parsed PDF file
- PDF reader: handle multi-page PDFs (#71)
- PDF reader: fix parsing of XRef streams
- PDF reader: fix writing of parsed files

10.10 Release 0.2.1 (2016-08-18)

New Features:

- optionally limit the width of large images and make use of this to simulate the Sphinx LaTeX builder behavior (#46)
- reStructuredText/Sphinx: support for images with hyperlinks (#49)
- record the styled page numbers in the PDF as page labels (#41)
- unsupported Python versions: prevent installation where possible (sdist) or exit on import (wheel)
- support Python 3.6

Bugfixes:

- make StyleSheet objects picklable so the Sphinx builder's rinoh_stylesheet option can actually be used
- Fix #47: ClassNotFound exception in Literal_Block.lexer_getter()
- Fix #45: Images that don't fit are still placed on the page
- don't warn about duplicate style matches that resolve to the same style

10.11 Release 0.2.0 (2016-08-10)

Styling:

- generate a style log (show matching styles) to help style sheet development
- keep_with_next style attribute: prevent splitting two flowables across pages
- stylesheets can be loaded from files in INI format
- check the type of attributes passed to styles
- source code highlighting using Pygments
- table of contents entries can be styled more freely
- allow hiding the section numbers of table of contents entries
- allow for custom chapter titles
- selectors can now also select based on document part/section
- various small tweaks to selectors and matchers
- various fixes relating to style sheets

Templates:

- configurable standard document templates: article and book
- a proper infrastructure for creating custom document templates
- support for left/right page templates
- make the Article template more configurable
- pages now have background, content and header/footer layers

- support for generating an index
- make certain strings configurable (for localization, for example)

Frontends:

- Sphinx: interpret the LaTeX configuration variables if the corresponding rinohtype variable is not set
- Sphinx: roughly match the LaTeX output (document template and style sheet)
- added a CommonMark frontend based on recommonmark
- added basic ePUB and DocBook frontends
- XML frontends: fix whitespace handling
- frontends now return generators yielding flowables (more flexible)

Command-line Renderer (rinoh):

- allow specifying a template and style sheet
- automatically install typefaces used in the style sheet from PyPI

Fonts:

- typefaces are discovered/loaded by entry point
- more complete support for OpenType fonts
- fix support for the 14 base Type 1 fonts

Images:

- more versatile image sizing: absolute width/height & scaling
- allow specifying the baseline for inline images
- several fixes in the JPEG reader

Miscellaneous:

- reorganize the Container class hierarchy
- fixes in footnote handling
- drop Python 3.2 support (3.3, 3.4 and 3.5 are supported)

10.12 Release 0.1.3 (2015-08-04)

- recover from the slow rendering speed caused by a bugfix in 0.1.2 (thanks to optimized element matching in the style sheets)
- other improvements and bugfixes related to style sheets

10.13 Release 0.1.2 (2015-07-31)

- much improved Sphinx support (we can now render the Sphinx documentation)
- more complete support for reStructuredText (docutils) elements
- various fixes related to footnote placement
- page break option when starting a new section
- fixes in handling of document sections and parts
- improvements to section/figure/table references
- native support for PNG and JPEG images (drops PIL/Pillow requirement, but adds PurePNG 0.1.1 requirement)
- new 'sphinx' stylesheet used by the Sphinx builder (~ Sphinx LaTeX style)
- restores Python 3.2 compatibility

10.14 Release 0.1.1 (2015-04-12)

First preview release

Index