RinohType

Brecht Machiels

December 18, 2015

Table of Contents

1 Introduction

1.1 Design Concepts

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.

Style sheets

The CSS-inspired style sheets determine the look of individual document elements.

Structured Input

RinohType renders a document from a document tree that does not describe any style aspects but only describes semantics. The style sheet maps specific style properties to the elements in this document tree. The document tree can be generated from a structured document format such as reStructuredText and DocBook using one of the included parsers, or it can be built manually.

As RinohType is implemented as a Python package, it doubles as a high-level PDF library. It's modular design makes it easy to to customize and extend for specific applications. Because RinohType's source code is open, all of its internals can be inspected and even modified, making it extremely customizable.

One RinohType's key design concepts is to limit the core's size to keep things simple but make it easy to build extensions. Currently the core of RinohType (excluding frontends, backends and font parsers) consists of less than 4000 lines.

1.2 Usage Examples

RinohType supports three modes of operation. These are discussed in more detail in Quickstart guide.

1.2.1 reStructuredText Renderer

RinohType comes with a command-line tool `rinoh` that can render reStructuredText documents. Rendering the reStructuredText demonstration article demo.txt (using the standard article template and style sheet) generates `demo.pdf`.

1.2.2 Sphinx Builder

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

1.2.3 High-level PDF library

RinohType can be used in a Python application to generate PDF documents. This basically involves three choices:

1) You can use one of the **document templates** included with RinohType and optionally customize it to your needs. Or you can create a custom template from scratch.
2) Choose to start from one of included **style sheets** or roll your own.
3) The **document tree** can be parsed from a structured document format such as reStructuredText or built manually using building blocks provided by RinohType. Both 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.

2 Installation

RinohType supports Python 3.2 and up. Depending on demand, it might be back-ported to Python 2.7, however[*].

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 Sphinx Builder in the Quickstart guide on how to render Sphinx documents with RinohType.

2.1 Dependencies

For parsing reStructuredText documents RinohType depends on docutils. For parsing PNG images the pure-Python PurePNG package is required. pip takes care of these requirements automatically when you install RinohType.

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

* Be sure to contact us if you are interested in running RinohType on Python 2.7.

* Be sure to contact us if you are interested in running RinohType on Python 2.7.

3 Quickstart

This section gets you started quickly, discussing each of the three modes of operation introduced in Introduction. Additionally, the basics of style sheets and document templates are explained.

3.1 reStructuredText Renderer

Installing RinohType places the `rinoh` script in the `PATH`. This can be used to render reStructuredText documents such as demo.txt:

```
rinoh demo.txt
```

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

At this moment, `rinoh` does not yet accept many command-line options. It always renders the reStructuredText document using the article template. You can however specify the paper size using the `--paper` command line argument.

3.2 Sphinx Builder

To use RinohType to render Sphinx documents, you need to adjust the Sphinx project's `conf.py`:

1) add `rinoh.frontend.sphinx` to the `extensions` list, and
2) set the `rinoh_documents` configuration option:

   ```
   rinoh_documents = [('index',            # top-level file (index.rst)
                       'target',           # output (target.pdf)
                       'Document Title',   # document title
                       'John A. Uthor')]   # document author
   ```

3) (optional) you can customize some aspects of the document template such as margins and headers and footer text by specifying `rinoh_options`:

   ```
   from rinoh.dimension import CM
   from rinoh.reference import Variable, PAGE_NUMBER, NUMBER_OF_PAGES
   from rinoh.text import Tab
   from rinoh.float import InlineImage

   footer_text = (InlineImage('images/company_logo.pdf')
                  + Tab() + Tab()
                  + Variable(PAGE_NUMBER) + ' / ' + Variable(NUMBER_OF_PAGES))

   rinoh_options = dict(header_text=Tab() + project,
                        footer_text=footer_text,
                        page_horizontal_margin=3*CM,
                        page_vertical_margin=4.5*CM)
   ```

   The Sphinx builder uses the `Book` document template†. `rinoh_options` are passed to `BookOptions`, so see its documentation for details.

4) now we can select the *rinoh* builder when building the documentation:

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

† This will be configurable in the future.

3.3 High-level PDF library

The most basic way to use the RinohType package is to hook up an included parser, a style sheet and a document template:

```
from rinoh.paper import A4
from rinoh.backend import pdf
from rinoh.frontend.rst import ReStructuredTextParser

from rinohlib.stylesheets.sphinx import styles as STYLESHEET
from rinohlib.templates.article import Article, ArticleOptions


# the parser builds a RinohType document tree
parser = ReStructuredTextParser()
with open('my_document.rst') as rst_file:
    document_tree = parser.parse(rst_file)

# customize the article template
article_options = ArticleOptions(page_size=A4, columns=2,
                                 table_of_contents=True,
                                 stylesheet=STYLESHEET)

# render the document to 'my_document.pdf'
document = Article(document_tree, options=article_options,
                   backend=pdf)
document.render('my_document')
```

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

The document tree returned by the `ReStructuredTextParser` in the example above can be easily built manually. *document_tree* is simply a list of `Flowable`s. These flowables can have children flowables. These in turn can also have children, and so on; together they form a tree.

Here is an example document tree of a short article:

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

It is obvious this type of content is best parsed from a structured document file such as reStructuredText or XML. Manually building a document tree is well suited for short, custom documents however. Please refer to the invoice example for details.

3.4 Style Sheets

A RinohType style sheet is defined in a Pyton source file, as an instance of the `StyleSheet` class. For each document element, the style sheet object registers a list of style properties.

This is similar to how HTML's cascading style sheets work. In RinohType however, style properties are assigned to document elements by means of a descriptive label for the latter instead of a selector. RinohType also makes use of selectors, but these are collected in a `StyledMatcher`. Unless you are using RinohType as a PDF library to creating custom documents, the default matcher should cover your needs.

3.4.1 Building on 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 example creates a new style sheet based on the Sphinx stylesheet included with RinohType. The style sheet redefines the style for emphasized text, displaying it in a bold instead of italic font.

```
from rinoh.dimension import PT
from rinoh.font.style import BOLD
from rinohlib.stylesheets.sphinx import styles

my_style_sheet = StyleSheet('My Style Sheet', base=styles)

my_style_sheet('emphasis', font_weight=BOLD)
```

Here, the new new style definition completely replaces the style definition contained in the Sphinx style sheet. It is also possible to override only part of the style definition. The following style definition changes only the item spacing between enumerated list items. All other style properties (such as the left margin and the item numbering format) remain unchanged.

```
my_style_sheet('enumerated list', base=styles['default'],
               flowable_spacing=3*PT)
```

3.4.2 Starting with a clean slate

Instantiating a new style sheet without passing it a base style sheet creates an independent style sheet. You need to specify the `StyledMatcher` to use in this case.

```
from rinoh.dimension import PT
from rinoh.font.style import BOLD
from rinohlib.stylesheets.sphinx import styles
from rinohlib.stylesheets.matcher import matcher

independent_style_sheet = StyleSheet('My Independent Style Sheet',
                                     matcher=matcher)
```

If a style definition for a particular document element is missing, the default values for its style properties are used.

3.5 Documument Templates

As with style sheets, you can choose to make use of the templates provided by RinohType or you can create a custom template from scratch. This section only covers the former. For an example of how to create a custom template, see the invoice example.

RinohType includes two document templates; `Article` and `Book`. Theese templates can be customized by passing an `ArticleOptions` or `BookOptions` instance as *options* on template instantiation respectively. Both these classes derive from `DocumentOptions` and thus accept the options offered by it:

class `rinohlib.templates.base.`**`DocumentOptions`** (***options*)

Collects options to customize a `DocumentTemplate`. Options are specified as keyword arguments (*options*) matching the class's attributes.

`columns`

Description	The number of columns for the body text (`int`)
Default	1

`footer_text`

Description	The text to place in the page footer (`MixedStyledText`)
Default	$PAGE_NUMBER/$NUMBER_OF_PAGES

`header_text`

Description	The text to place in the page header (`MixedStyledText`)
Default	$SECTION_NUMBER(1) $SECTION_TITLE(1)

`page_horizontal_margin`

Description	The margin size on the left and the right of the page (`DimensionBase`)
Default	85.04pt

`page_orientation`

Description	The orientation of pages in the document (`PageOrientation`)
Default	portrait

`page_size`

Description	The format of the pages in the document (`Paper`)
Default	A4

`page_vertical_margin`

Description	The margin size on the top and bottom of the page (`DimensionBase`)
Default	85.04pt

`show_author`

Description	Show or hide the document's author (`bool`)
Default	True

`show_date`

Description	Show or hide the document's date (`bool`)
Default	True

`stylesheet`

Description	The stylesheet to use for styling document elements (`StyleSheet`)

Default StyleSheet(Sphinx)

The `Article` and `Book` templates also have some specific options:

class `rinohlib.templates.article.`**`ArticleOptions`** (***options*)

`abstract_location`

Description	Where to place the abstract (`AbstractLocation`)
Default	front_matter

`table_of_contents`

Description	Show or hide the table of contents (`bool`)
Default	True

class `rinohlib.templates.book.`**`BookOptions`** (***options*)

`extra`

Description	Extra text to include on the title page below the title (`MixedStyledText`)
Default	None

† This will be configurable in the future.

4 Advanced Topics

This sections serves as a reference for various building blocks making up RinohType. The information presented here is useful for those who want to learn how element styling works in RinohType, which is helpful when creating custom style sheets.

4.1 Flowables and Inline Elements

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 number or bullet)
     |    |    \- 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 other 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 eventual style of the words in this paragraph is determined by the applied style sheet. Read more about this 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 discussed in the next section.

4.2 Style Sheets

RinohType's style sheets are heavily inspired by CSS, but add some functionality that CSS lacks. Similar to CSS, RinohType makes use of so-called *selectors* to select document elements (flowables or inline elements) 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. The actual *style sheets* assign 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.

4.2.1 Selectors

Selectors in RinohType always select elements of a particular type. The **class** of a document element is also a selector for all instances of the class (and its subclasses). This selector 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 (the list item number or symbol):

```
ListItem / Paragraph
```

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

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

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

Similar to HTML/CSS's *class* attribute, `Styled` elements can have a **style** attribute which can be specified when constructing a selector. This one selects all styled text elements with the *emphasis* style, for example:

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

The `like()` method can also match **arbitrary attributes** of elements. 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 slice object that is used for extended indexing. Indexing a list `lst[slice(0, None, 2)]` is equivalent to `lst[0::2]`.

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

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

wins over:

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

since it matches two elements instead of just one.

Specificity is represented as a 3-tuple. The three elements represent the number of style, attributes and class matches. Here are some selectors along with their specificity:

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

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

4.2.2 Matchers

At the most basic level, a `StyledMatcher` is a dictionary that maps descriptions 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 styled matcher which defines labels for all common elements in documents:

```
from rinohlib.stylesheets.matcher import matcher
```

4.2.3 Style Sheets

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

```
styles = StyleSheet('IEEE', matcher=matcher)
...
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. The style class is automatically determined from the selector, so it is possible to simply pass the style properties to the style sheet.

Variables

Variables can be used for values that are used in multiple style definitions. This example declares a variable `fonts` to allow easily changing the fonts in a style sheet:

```
from rinoh.font import TypeFamily

from rinohlib.fonts.texgyre.pagella import typeface as palatino
from rinohlib.fonts.texgyre.cursor import typeface as courier
from rinohlib.fonts.texgyre.heros import typeface as helvetica

styles.variables['font'] = TypeFamily(serif=times,
                                                sans=helvetica,
                                                mono=courier)
...
styles('monospaced',
       typeface=Var('font_family').mono,
       font_size=9*PT,
       hyphenate=False,
       ligatures=False)
...
```

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

Style Property Resolution

The style system makes a distinction between text (inline) elements and flowables with respect to how property 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. That can be another `StyledText` instance, or a `Paragraph`. If that style definition neither defines the style property, the lookup proceeds recursively, moving up in the document tree.

For **flowables**, there is no fall-back to the parent's style by default. A base style can be explicitly specified however. If a style property is not present in a particular style definition, it is looked up in the base style.

This can also 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` property:

```
styles('heading level 1',
       typeface=Var('ieee_family').serif,
       font_weight=REGULAR,
       font_size=10*PT,
       small_caps=True,
       justify=CENTER,
       line_spacing=FixedSpacing(12*PT),
       space_above=18*PT,
       space_below=6*PT,
       number_format=ROMAN_UC,
       label_suffix='.' + FixedWidthSpace())

styles('unnumbered heading level 1',
       base='heading level 1',
       number_format=None)
```

When a value for a particular style property 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.

For both text elements and flowables, it is possible to override the default behavior of falling back to the parent's style or not. For `TextStyle` styles, setting `base` to `None` or another `TextStyle` prevents fallback to the parent element's style. For flowables, `base` can be set to `PARENT_STYLE` to enable fallback, but this requires that the current element type is the same or a subclass of the parent type, so it is not recommended.

Extending a Style Sheet

A style sheet can be extended by defining a new style sheet that references it as a base:

```
from rinohlib.stylesheets.sphinx import stylesheet

my_style_sheet = StyleSheet('my style', base=stylesheet)
```

The new stylesheet can override styles defined in the base style sheet. The following redefines the `emphasis` style to display emphasized text in a bold font:

my_style_sheet('emphasis', font_weight=BOLD)

Variables can also be overridden. This overrides the `fonts` variable in order to replace the serif font defined in the Sphinx style sheet (Palatino) with Times:

```
from rinohlib.fonts.texgyre.termes import typeface as times
from rinohlib.fonts.texgyre.cursor import typeface as courier
from rinohlib.fonts.texgyre.heros import typeface as helvetica

my_style_sheet.variables['fonts'] = TypeFamily(serif=times,
                                               sans=helvetica,
                                               mono=courier)
```

The variable's new value also affects styles defined in the base style sheet.

The new style sheet can optionally be passed a `StyledMatcher` to define new styles. This is useful when you want to have custom markup in your document, such as custom roles or directives are used in a reStructuredText document. For example, the following defines a custom style to apply to text that is tagged with the `acronym` role:

```
my_matcher = StyledMatcher()
my_matcher['acronym'] = StyledText.like(classes=['acronym'])

my_style_sheet = StyleSheet('my style', base=stylesheet,
                            matcher=my_matcher)
my_style_sheet('acronym', small_caps=True)
```

5 API Documentation

5.1 Flowables

5.1.1 Base Class for Flowables

class `rinoh.flowable.`**`Flowable`** (*id=None*, *style=None*, *parent=None*)

An element that can be 'flowed' into a `Container`. A flowable can adapt to the width of the container, or it can horizontally align itself in the container.

class `rinoh.flowable.`**`FlowableStyle`** (*base=None*, ***attributes*)

The `Style` for `Flowable` objects. It has the following attributes:

- *space_above*: Vertical space preceding the flowable (`Dimension`)
- *space_below*: Vertical space following the flowable (`Dimension`)
- *margin_left*: Left margin (class:*Dimension*).
- *margin_right*: Right margin (class:*Dimension*).
- ***horizontal_align*: Alignment of the rendered flowable between the left**

  and right margins (*LEFT*, *CENTER* or *RIGHT*).

class `rinoh.flowable.`**`FlowableState`** (*_initial=True*)

Stores a `Flowable`'s rendering state, which can be copied. This enables saving the rendering state at certain points in the rendering process, so rendering can later be resumed at those points, if needed.

5.1.2 Flowables that Do Not Render Anything

These flowables do not place anything on the page. All except `DummyFlowable` do have side-effects however. Some of these side-effects affect the rendering of the document in an indirect way.

class `rinoh.flowable.`**`DummyFlowable`** (*parent=None*)

class `rinoh.flowable.`**`SetMetadataFlowable`** (*parent=None*, ***metadata*)

class `rinoh.flowable.`**`WarnFlowable`** (*message*, *parent=None*)

class `rinoh.flowable.`**`PageBreak`** (*break_type*)

5.1.3 Labeled Flowables

class `rinoh.flowable.`**`LabeledFlowable`** (*label*, *flowable*, *id=None*, *style=None*, *parent=None*)

class `rinoh.flowable.`**`LabeledFlowableStyle`** (*base=None*, ***attributes*)

5.1.4 Grouping Flowables

class `rinoh.flowable.`**`GroupedFlowables`** (*id=None, style=None, parent=None*)

class `rinoh.flowable.`**`GroupedFlowablesStyle`** (*base=None, **attributes*)

class `rinoh.flowable.`**`GroupedFlowablesState`** (*flowables, first_flowable_state=None, _initial=True*)

class `rinoh.flowable.`**`StaticGroupedFlowables`** (*flowables, id=None, style=None, parent=None*)

class `rinoh.flowable.`**`GroupedLabeledFlowables`** (*id=None, style=None, paren-t=None*)

5.1.5 Horizontally Aligned Flowables

class `rinoh.flowable.`**`HorizontallyAlignedFlowable`** (*id=None, style=None, paren-t=None*)

class `rinoh.flowable.`**`HorizontallyAlignedFlowableStyle`** (*base=None, **at-tributes*)

class `rinoh.flowable.`**`HorizontallyAlignedFlowableState`** (*_initial=True*)

5.1.6 Floating Flowables

class `rinoh.flowable.`**`Float`** (*flowable, style=None, parent=None*)

Transform a `Flowable` into a floating element. A floating element or 'float' is not flowed into its designated container, but is forwarded to another container pointed to by the former's `Container.float_space` attribute.

This is typically used to place figures and tables at the top or bottom of a page, instead of in between paragraphs.

5.2 Paragraph

class `rinoh.paragraph.`**`ParagraphBase`** (*id=None, style=None, parent=None*)

A paragraph of mixed-styled text that can be flowed into a `Container`.

> **`render`** (*container, descender, state=None*)
>
> Typeset the paragraph onto *container*, starting below the current cursor position of the container. *descender* is the descender height of the preceeding line or *None*. When the end of the container is reached, the rendering state is preserved to continue setting the rest of the paragraph when this method is called with a new container.
>
> **`style_class`**
>
> alias of `ParagraphStyle`

class `rinoh.paragraph.`**`Paragraph`** (*text_or_items, id=None, style=None, parent=None*)

class `rinoh.paragraph.`**`ParagraphStyle`** (*base=None, **attributes*)

The `Style` for `Paragraph` objects. It has the following attributes:

- *indent_first*: Indentation of the first line of text (class:*Dimension*).
- ***line_spacing*: Spacing between the baselines of two successive lines of**
  text (`LineSpacing`).
- ***justify*: Alignment of the text to the margins (`LEFT`,**
  `RIGHT`, `CENTER` or `BOTH`).
- *tab_stops*: The tab stops for this paragraph (list of `TabStop`).

class `rinoh.paragraph.`**`ParagraphState`** (*words*, *nested_flowable_state=None*, *_first_word=None*, *_initial=True*)

5.2.1 Inline Elements

class `rinoh.text.`**`StyledText`** (*style=None*, *parent=None*)

Base class for text that has a `TextStyle` associated with it.

`height` (*document*)

Font size after super/subscript size adjustment.

`is_script` (*document*)

Returns *True* if this styled text is super/subscript.

`script_level` (*document*)

Nesting level of super/subscript.

`spans` ( )

Generator yielding all spans in this styled text, one item at a time (used in typesetting).

`style_class`

alias of `TextStyle`

`y_offset` (*document*)

Vertical baseline offset (up is positive).

class `rinoh.text.`**`SingleStyledText`** (*text*, *style=None*, *parent=None*)

Styled text where all text shares a single `TextStyle`.

`font` (*document*)

The `Font` described by this single-styled text's style.

If the exact font style as described by the *font_weight*, *font_slant* and *font_width* style attributes is not present in the *typeface*, the closest font available is returned instead, and a warning is printed.

`split` (*container*)

Yield the words and spaces in this single-styled text.

class `rinoh.text.`**`MixedStyledText`** (*text_or_items*, *style=None*, *parent=None*)

Concatenation of `StyledText` objects.

`append` (*item*)

Append *item* (`StyledText` or `str`) to the end of this mixed-styled text.

The parent of *item* is set to this mixed-styled text.

`spans` (*document*)

Recursively yield all the `SingleStyledText` items in this mixed-styled text.

5.2.2 Styling Properties

Line Spacing

class `rinoh.paragraph.`**`LineSpacing`**

Base class for line spacing types. Line spacing is defined as the distance between the baselines of two consecutive lines.

`advance` (*line*, *last_descender*, *document*)

Return the distance between the descender of the previous line and the baseline of the current line.

class `rinoh.paragraph.`**`DefaultSpacing`**

The default line spacing as specified by the font.

class `rinoh.paragraph.`**`ProportionalSpacing`** (*factor*)

Line spacing proportional to the line height.

class `rinoh.paragraph.`**`FixedSpacing`** (*pitch*, *minimum=<rinoh.paragraph.ProportionalSpacing object>*)

Fixed line spacing, with optional minimum spacing.

class `rinoh.paragraph.`**`Leading`** (*leading*)

Line spacing determined by the space in between two lines.

Tabulation

class `rinoh.paragraph.`**`TabStop`** (*position*, *align='left'*, *fill=None*)

Horizontal position for aligning text of successive lines.

`get_position` (*line_width*)

Return the absolute position of this tab stop.

class `rinoh.paragraph.`**`Paragraph`** (*text_or_items*, *id=None*, *style=None*, *parent=None*)

class `rinoh.paragraph.`**`Paragraph`** (*text_or_items*, *id=None*, *style=None*, *parent=None*)

5.2.3 Rendering Internals

class `rinoh.paragraph.`**`GlyphAndWidth`** (*glyph*, *width*)

class `rinoh.paragraph.`**`GlyphsSpan`** (*span*, *word_to_glyphs*)

5.2.4 Miscellaneous Internals

class `rinoh.paragraph.`**`HyphenatorStore`**

6 Release History

6.1 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

6.2 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

6.3 Release 0.1.1 (2015-04-12)

First preview release