34.1. `formatter` — Generic output formatting¶

This module supports two interface definitions, each with multiple implementations. The *formatter* interface is used by the HTMLParser class of the htmllib module, and the *writer* interface is required by the formatter interface.

Formatter objects transform an abstract flow of formatting events into specific output events on writer objects. Formatters manage several stack structures to allow various properties of a writer object to be changed and restored; writers need not be able to handle relative changes nor any sort of "change back" operation. Specific writer properties which may be controlled via formatter objects are horizontal alignment, font, and left margin indentations. A mechanism is provided which supports providing arbitrary, non-exclusive style settings to a writer as well. Additional interfaces facilitate formatting events which are not reversible, such as paragraph separation.

Writer objects encapsulate device interfaces. Abstract devices, such as file formats, are supported as well as physical devices. The provided implementations all work with abstract devices. The interface makes available mechanisms for setting the properties which formatter objects manage and inserting data into the output.

34.1.1. The Formatter Interface¶

Interfaces to create formatters are dependent on the specific formatter class being instantiated. The interfaces described below are the required interfaces which all formatters must support once initialized.

One data element is defined at the module level:

formatter.AS_IS¶

Value which can be used in the font specification passed to the `push_font()` method described below, or as the new value to any other `push_property()` method. Pushing the AS_IS value allows the corresponding `pop_property()` method to be called without having to track whether the property was changed.

The following attributes are defined for formatter instance objects:

formatter.writer¶

The writer instance with which the formatter interacts.

formatter.end_paragraph(*blanklines*)¶

Close any open paragraphs and insert at least *blanklines* before the next paragraph.

formatter.add_line_break()¶

Add a hard line break if one does not already exist. This does not break the logical paragraph.

formatter.add_hor_rule(**args*, ***kw*)¶

Insert a horizontal rule in the output. A hard break is inserted if there is data in the current paragraph, but the logical paragraph is not broken. The arguments and keywords are passed on to the writer's `send_line_break()` method.

formatter.add_flowing_data(*data*)¶

Provide data which should be formatted with collapsed whitespace. Whitespace from preceding and successive calls to add_flowing_data() is considered as well when the whitespace collapse is performed. The data which is passed to this method is expected to be word-wrapped by the output device. Note that any word-wrapping still must be performed by the writer object due to the need to rely on device and font information.

formatter.add_literal_data(*data*)¶

Provide data which should be passed to the writer unchanged. Whitespace, including newline and tab characters, are considered legal in the value of *data*.

formatter.add_label_data(*format*, *counter*)¶

Insert a label which should be placed to the left of the current left margin. This should be used for constructing bulleted or numbered lists. If the *format* value is a string, it is interpreted as a format specification for *counter*, which should be an integer. The result of this formatting becomes the value of the label; if *format* is not a string it is used as the label value directly. The label value is passed as the only argument to the writer's `send_label_data()` method. Interpretation of non-string label values is dependent on the associated writer.

Format specifications are strings which, in combination with a counter value, are used to compute label values. Each character in the format string is copied to the label value, with some characters recognized to indicate a transform on the counter value. Specifically, the character `'1'` represents the counter value formatter as an Arabic number, the characters `'A'` and `'a'` represent alphabetic representations of the counter value in upper and lower case, respectively, and `'I'` and `'i'` represent the counter value in Roman numerals, in upper and lower case. Note that the alphabetic and roman transforms require that the counter value be greater than zero.

`formatter.flush_softspace()`¶

Send any pending whitespace buffered from a previous call to `add_flowing_data()` to the associated writer object. This should be called before any direct manipulation of the writer object.

`formatter.push_alignment(`*align*`)`¶

Push a new alignment setting onto the alignment stack. This may be `AS_IS` if no change is desired. If the alignment value is changed from the previous setting, the writer's `new_alignment()` method is called with the *align* value.

`formatter.pop_alignment()`¶

Restore the previous alignment.

`formatter.push_font(`*(size*, *italic*, *bold*, *teletype)*`)`¶

Change some or all font properties of the writer object. Properties which are not set to `AS_IS` are set to the values passed in while others are maintained at their current settings. The writer's `new_font()` method is called with the fully resolved font specification.

`formatter.pop_font()`¶

Restore the previous font.

`formatter.push_margin(`*margin*`)`¶

Increase the number of left margin indentations by one, associating the logical tag *margin* with the new indentation. The initial margin level is `0`. Changed values of the logical tag must be true values; false values other than `AS_IS` are not sufficient to change the margin.

`formatter.pop_margin()`¶

Restore the previous margin.

`formatter.push_style(`**styles*`)`¶

Push any number of arbitrary style specifications. All styles are pushed onto the styles stack in order. A tuple representing the entire stack, including `AS_IS` values, is passed to the writer's `new_styles()` method.

`formatter.pop_style(`[*n=1*]`)`¶

Pop the last *n* style specifications passed to `push_style()`. A tuple representing the revised stack, including `AS_IS` values, is passed to the writer's `new_styles()` method.

`formatter.set_spacing(`*spacing*`)`¶

Set the spacing style for the writer.

`formatter.assert_line_data(`[*flag=1*]`)`¶

Inform the formatter that data has been added to the current paragraph out-of-band. This should be used when the writer has been manipulated directly. The optional *flag* argument can be set to false if the writer manipulations produced a hard line break at the end of the output.

34.1.2. Formatter Implementations¶

Two implementations of formatter objects are provided by this module. Most applications may use one of these classes without modification or subclassing.

class `formatter.NullFormatter(`[*writer*]`)`¶

A formatter which does nothing. If *writer* is omitted, a `NullWriter` instance is created. No methods of the writer are called by `NullFormatter` instances. Implementations should inherit from this class if implementing a writer interface but don't need to inherit any implementation.

class `formatter.AbstractFormatter(`*writer*`)`¶

The standard formatter. This implementation has demonstrated wide applicability to many writers, and may be used directly in most circumstances. It has been used to implement a full-featured World Wide Web browser.

34.1.3. The Writer Interface¶

Interfaces to create writers are dependent on the specific writer class being instantiated. The interfaces described below are the required interfaces which all writers must support once initialized. Note that while most applications can use the `AbstractFormatter` class as a formatter, the writer must typically be provided by the application.

`writer.flush()`¶

Flush any buffered output or device control events.

`writer.new_alignment(`*align*`)`¶

Set the alignment style. The *align* value can be any object, but by convention is a string or `None`, where `None` indicates that the writer's "preferred" alignment should be used. Conventional *align* values are `'left'`, `'center'`, `'right'`, and `'justify'`.

`writer.new_font(`*font*`)`¶

Set the font style. The value of *font* will be `None`, indicating that the device's default font should be used, or a tuple of the form `(size, italic, bold, teletype)`. Size will be a string indicating the size of font that should be used; specific strings and their interpretation must be defined by the application. The *italic*, *bold*, and *teletype* values are Boolean values specifying which of those font attributes should be used.

`writer.new_margin(`*margin*, *level*`)`¶

Set the margin level to the integer *level* and the logical tag to *margin*. Interpretation of the logical tag is at the writer's discretion; the only restriction on the value of the logical tag is that it not be a false value for non-zero values of *level*.

`writer.new_spacing(`*spacing*`)`¶

Set the spacing style to *spacing*.

`writer.new_styles(`*styles*`)`¶

Set additional styles. The *styles* value is a tuple of arbitrary values; the value `AS_IS` should be ignored. The *styles* tuple may be interpreted either as a set or as a stack depending on the requirements of the application and writer implementation.

`writer.send_line_break`()¶

Break the current line.

`writer.send_paragraph`(*blankline*)¶

Produce a paragraph separation of at least *blankline* blank lines, or the equivalent. The *blankline* value will be an integer. Note that the implementation will receive a call to `send_line_break()` before this call if a line break is needed; this method should not include ending the last line of the paragraph. It is only responsible for vertical spacing between paragraphs.

`writer.send_hor_rule`(**args*, ***kw*)¶

Display a horizontal rule on the output device. The arguments to this method are entirely application- and writer-specific, and should be interpreted with care. The method implementation may assume that a line break has already been issued via `send_line_break()`.

`writer.send_flowing_data`(*data*)¶

Output character data which may be word-wrapped and re-flowed as needed. Within any sequence of calls to this method, the writer may assume that spans of multiple whitespace characters have been collapsed to single space characters.

`writer.send_literal_data`(*data*)¶

Output character data which has already been formatted for display. Generally, this should be interpreted to mean that line breaks indicated by newline characters should be preserved and no new line breaks should be introduced. The data may contain embedded newline and tab characters, unlike data provided to the `send_formatted_data()` interface.

`writer.send_label_data`(*data*)¶

Set *data* to the left of the current left margin, if possible. The value of *data* is not restricted; treatment of non-string values is entirely application- and writer-dependent. This method will only be called at the beginning of a line.

34.1.4. Writer Implementations¶

Three implementations of the writer object interface are provided as examples by this module. Most applications will need to derive new writer classes from the `NullWriter` class.

class `formatter.NullWriter`¶

A writer which only provides the interface definition; no actions are taken on any methods. This should be the base class for all writers which do not need to inherit any implementation methods.

class `formatter.AbstractWriter`¶

A writer which can be used in debugging formatters, but not much else. Each method simply announces itself by printing its name and arguments on standard output.

class `formatter.DumbWriter`([*file*[, *maxcol=72*]])¶

Simple writer class which writes output on the file object passed in as *file* or, if *file* is omitted, on standard output. The output is simply word-wrapped to the number of columns specified by *maxcol*. This class is suitable for reflowing a sequence of paragraphs.

Table Of Contents

Previous topic

Next topic

This Page

- Show Source

Navigation