linotype Documentation

Release 0.2.1

Garrett Powell

Feb 21, 2018

Contents:

CHAPTER 1

API

linotype.**ansi_format**(*fg=None*, *bg=None*, *bold=False*, *underline=False*) → typing.Tuple[str, str]

Get the appropriate ANSI escape sequences based on input.

Not all features are supported on all terminals.

Parameters

- **fg** – The foreground color specification. This can be the name of one of the eight ANSI colors, an integer in the range 0-255 or a CSS-style hex value. 'None' means no formatting.
- **bg** – The background color specification. This can be the name of one of the eight ANSI colors, an integer in the range 0-255 or a CSS-style hex value. 'None' means no formatting.
- **bold** – Make the text bold.
- **underline** – Make the text underlined.

Raises ValueError – The given color spec was unrecognized.

Returns A tuple containing the starting and ending ANSI escape sequences.

class linotype.**DefStyle**

Styles for definition items.

PARAGRAPH, P

Display the message as a paragraph on a separate line from the term and argument string.

INLINE, I

Display the message on the same line as the term and argument string. Use a hanging indent if the message is too long.

ALIGNED, A

Display the message on the same line as the term and argument string and align the message with those of all other definitions that belong to the same parent item and have the style ALIGNED. Use a hanging indent if the message is too long.

OVERFLOW, O

Display the message on a separate line from the term and argument string and align the message with those

of all other definitions that belong to the same parent item and have a style of ALIGNED. Use a hanging indent if the message is too long.

class linotype.**Formatter**(*max_width=79*, *auto_width=True*, *indent_spaces=4*, *def_gap=2*, *def_style=<DefStyle.PARAGRAPH: 1>*, *auto_markup=True*, *manual_markup=True*, *visible=True*, *strong=('x1b[1m'*, *'x1b[0m')*, *em=('x1b[4m'*, *'x1b[0m')*) → None

Control how the text output is formatted.

max_width
The maximum number of columns at which to wrap text in the text output.

auto_width
Wrap the text to the size of the terminal.

indent_spaces
The number of spaces to increase/decrease the indent level by for each level.

def_gap
The minimum number of spaces to leave between the argument string and message of each definition when they are on the same line.

def_style
A DefStyle instance representing the style of definition to use.

auto_markup
Automatically apply 'strong' and 'emphasized' formatting to certain text in the output.

manual_markup
Parse reST 'strong' and 'emphasized' inline markup.

visible
Make the text visible in the output.

strong
A 2-tuple containing the strings to print before and after strings marked up as 'strong'. The default is ANSI bold.

em
A 2-tuple containing the strings to print before and after strings marked up as 'emphasized'. The default is ANSI underlined.

class linotype.**Item**(*formatter=<linotype.items.Formatter object>*) → None

An item to be displayed in the output.

This class allows for formatting text output consisting of a tree of "items". There are multiple types of items to choose from, and every item can contain other items which are indented relative to the parent item. Items at the same level are displayed at the order in which they were created.

This class is used to create a root-level item, and new child items can be created using its public methods. Every item has a Formatter instance which determines how it is formatted in the text output. Every item also has an ID which can be referenced in the Sphinx documentation or when formatting the text output.

Parameters **formatter** – The Formatter object for the item tree.

content
The content to display in the output.

formatter
The Formatter object for the item tree.

id
The item ID.

current_level
The current indentation level.

parent
The parent Item object.

_format_func
The function used for formatting the text output.

_current_indent
The number of spaces that the item is currently indented.

children
A list of all Item objects belonging to this item.

add_def(*term: str*, *args: str*, *message: str*, *formatter=None*, *item_id=None*) → linotype.items.Item
Add a definition item to be printed.

This item displays a formatted definition in one of multiple styles. The style is set by the Formatter instance. Definitions consist of a term, an argument string and a message, any of which can be blank.

Parameters

- **term** – The command, option, etc. to be defined. If auto markup is enabled, this is strong in the text output.
- **args** – The list of arguments for the thing being defined as a single string. If auto markup is enabled, arguments are emphasized in the text output. Arguments are consecutive strings of unicode word characters
- **message** – A description of the thing being defined, with arguments that appear in the argument string emphasized if auto markup is enabled.
- **formatter** – A Formatter instance for defining the formatting of the new item. If 'None,' it uses the formatter of its parent item.
- **item_id** – A unique ID for the item that can be referenced in the Sphinx documentation or when formatting the text output.

Returns The new Item object.

add_text(*text: str*, *formatter=None*, *item_id=None*) → linotype.items.Item
Add a text item to be printed.

This item displays the given text wrapped to the given width.

Parameters

- **text** – The text to be printed.
- **formatter** – A Formatter object for defining the formatting of the new item. If 'None,' it uses the formatter of its parent item.
- **item_id** – A unique ID for the item that can be referenced in the Sphinx documentation or when formatting the text output.

Returns The new Item object.

format(*levels=None*, *item_id=None*) → str
Print a tree of items.

Parameters

- **levels** – The number of levels of nested items to descend into.
- **item_id** – The ID of the root item. If 'None,' this defaults to the current item.

Returns The text output as a single string.

CHAPTER 2

Usage

Documentation in **linotype** consists of a tree of 'items' of which there are currently two types, *text* and *definitions*. Every item can contain other items which are indented relative to the parent item. The *linotype.Item* class is used to create a root-level item, and every *linotype.Item* object has public methods for creating new sub-items which in turn return a new *linotype.Item* object.

Every *linotype.Item* object accepts a *linotype.Formatter* instance which is used to define how items are formatted in the text output. Every item can optionally be assigned an ID that can be referenced in the **Sphinx** documentation or when formatting the text output. Here is an example that prints a simple help message:

```
from linotype import Item

def help_message():
    root_item = Item()

    usage = root_item.add_text("Usage:")
    usage.add_def(
        "todo", "[global_options] command [command_args]", "")

    return root_item

print(help_message().format())
```

Line wrapping, indentation, alignment and markup are all applied automatically according to attributes set in the *linotype.Formatter* object, so all text is passed into **linotype** as unformatted, single-line strings. Additionally, inline 'strong' and 'emphasized' markup can be applied manually using the reStructuredText syntax:

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

To use **linotype** with **Sphinx**, you must first add 'linotype.ext' to the list of **Sphinx** extensions in the *conf.py* file for your project:

```
extensions = ["linotype.ext"]
```

The documentation can be imported into your **Sphinx** documentation using the 'linotype' directive. It accepts the following options:

:function: The name of the function which returns a *linotype.Item* object.

:module: The name of the module containing the function.

:filepath: The path of the python file containing the function.

:item_id: The ID of an item in the tree returned by the function. The output is restricted to just this item and its children.

:children: Display the item's children but not the item itself.

:no_auto_markup: Do not automatically apply 'strong' and 'emphasized' formatting to the output.

:no_manual_markup: Do not parse 'strong' and 'emphasized' inline markup.

The options :module: and :filepath: are mutually exclusive. The options :function: and either :module: or :filepath: are required.

Using the 'linotype' directive, you can extend or replace parts of your help message. This allows you to add new content that appears in your **Sphinx** documentation but not in your text output. This is done on a per-item basis using a reStructuredText definition list, where the term is the ID of an item and the definition is the new content to use. You can also add classifiers, which change how the new content is incorporated.

These classifiers affect where the content is added:

@after The new content is added after the existing content. This is the default.

@before The new content is added before the existing content.

@replace The new content replaces the existing content.

These classifiers affect how markup is applied to the content:

@auto Markup is applied to the text automatically, and 'strong' and 'emphasized' inline markup can be applied manually. This is the default.

@rst Markup is not applied automatically, but any reStructuredText body or inline elements can be used. The new content starts in a separate paragraph.

Here is an example of a **Sphinx** source file using the directive:

```
.. linotype::
    :module: todo.cli
    :function: help_message

    add
        This content is added after the existing content for the item with
        the ID 'add.' Markup is applied automatically.

    add : @before : @rst
        This content is added before the existing content for the item with
        the ID 'add.' reStrcturedText elements can be used.

    check : @replace
        This content replaces the existing content for the item with the ID
        'check.' Markup is applied automatically.
```

CHAPTER 3

Examples

3.1 Add blank lines

A blank line can be added to your text output using a *text* item containing a newline character.

```
from linotype import Item

def help_message():
    root_item = Item()

    root_item.add_text("This line comes before the break.")
    root_item.add_text("\n")
    root_item.add_text("This line comes after the break.")

    return root_item
```

3.2 Configure markup in text output

The helper function *`linotype.ansi_format()`* can be used to generate ANSI escape sequences to configure the style of markup in the text output.

```
from linotype import ansi_format, Formatter, Item

def help_message():
    formatter = Formatter(
        strong=ansi_format(fg="red", bold=True),
        em=ansi_format(fg="green", bold=True))
    root_item = Item(formatter)

    root_item.add_text("This text has **strong** and *emphasized* markup.")

    return root_item
```

```

print(help_message().format())
```

3.3 Create two-column options lists

Many programs display command-line options in a two-column list with the options and arguments on the left and descriptions on the right. *Definition* items with the styles ALIGNED and OVERFLOW can be used for this purpose. The latter style is intended for options that are too long to otherwise fit.

```
from linotype import DefStyle, Formatter, Item

def help_message():
    aligned_formatter = Formatter(def_style=DefStyle.ALIGNED)
    overflow_formatter = Formatter(def_style=DefStyle.OVERFLOW)

    root_item = Item(aligned_formatter)

    root_item.add_def(
        "-q, --quiet", "", "Suppress all non-error output.")
    root_item.add_def(
        "-v, --verbose", "", "Increase verbosity.")
    root_item.add_def(
        "    --debug", "",
        "Print a full stack trace instead of an error message if an error "
        "occurs.")
    root_item.add_def(
        "    --from-file", "path",
        "Use the list from the file located at path.",
        formatter=overflow_formatter)

    return root_item

print(help_message().format())
```

This is what the output looks like:

```
-q, --quiet     Suppress all non-error output.
-v, --verbose   Increase verbosity.
    --debug     Print a full stack trace instead of an error message if an
                    error occurs.
    --from-file path
                Use the list from the file located at path.
```

3.4 Split message into sections

Instead of having your entire help message appear in one place in your **Sphinx** documentation, you may want to split it up into different sections. This can be accomplished by assigning item IDs.

```
from linotype import Item

def help_message():
    root_item = Item()
```

```

    usage = root_item.add_text("Usage:", item_id="usage")
    usage.add_def(
        "todo", "[global_options] command [command_args]", "")

    global_opts = root_item.add_text("Global Options:", item_id="global")
    global_opts.add_def(
        "-q, --quiet", "", "Suppress all non-error output.")

    return root_item

print(help_message().format())
```

This is what your **Sphinx** source file could look like:

```
SYNOPSIS
========
.. linotype::
    :module: todo.cli
    :function: help_message
    :item_id: usage
    :children:

GLOBAL OPTIONS
==============
.. linotype::
    :module: todo.cli
    :function: help_message
    :item_id: global
    :children:
```

3.5 Hide message details

To improve readability, you may want to only show certain details in your help message under certain circumstances. One example would be to have a main help message that displays an overview of all commands and then a separate help message with more details for each command. This can be accomplished by:

1. Limiting the number of levels of nested items to descend into (see *linotype.Item.format()*).
2. Conditionally making some items invisible via a *linotype.Formatter* class.
3. Creating a separate function for the per-command help messages.

The third method is shown below.

```
from linotype import Item

def main_help_message():
    root_item = Item()

    commands = root_item.add_text("Commands:")
    commands.add_def(
        "check", "[options] tasks...",
        "Mark one or more tasks as completed.")

    return root_item

```

```
def command_help_message():
    root_item = Item()

    check = root_item.add_def(
        "check", "[options] tasks...",
        "Mark one or more tasks as completed. These will appear hidden in "
        "the list.", item_id="check")
    check.add_def(
        "-r, --remove", "", "Remove the tasks from the list.")

    return root_item

if command:
    print(command_help_message().format(item_id=command))
else:
    print(main_help_message().format())
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

C

D

E

F

I

M

P

S

V