Enlighten Documentation

Release 1.12.0

Avram Lubkin

Sep 24, 2023

Contents

1 PIP **1**

2 RPM **3**
2.1 Fedora and EL8 (RHEL/CentOS) . . . 3

3 DEB **5**
3.1 Debian and Ubuntu . . . 5

4 Conda **7**

5 How to Use **9**
5.1 Managers . . . 9
5.2 Progress Bars . . . 9
5.3 Counters . . . 10
5.4 Status Bars . . . 10
5.5 Color . . . 11
5.6 Multicolored . . . 12
5.7 Additional Examples . . . 14
5.8 Customization . . . 14

6 Common Patterns **15**
6.1 Enable / Disable . . . 15
6.2 Context Managers . . . 15
6.3 Automatic Updating . . . 16
6.4 User-defined fields . . . 16
6.5 Human-readable numeric prefixes . . . 17
6.6 Manually Printing . . . 17

7 Frequently Asked Questions **19**
7.1 Why is Enlighten called Enlighten? . . . 19
7.2 Is Windows supported? . . . 19
7.3 Is Jupyter Notebooks Supported? . . . 19
7.4 Is PyCharm supported? . . . 19
7.5 Can you add support for _____ terminal? . . . 20
7.6 Why does `RuntimeError: reentrant call` get raised sometimes during a resize? . . . 20
7.7 Why isn't my progress bar displayed until `update()` is called? . . . 21

8 API Reference **23**
8.1 Classes 23
8.2 Functions 34
8.3 Constants 35

9 Overview **37**

Python Module Index **39**

Index **41**

CHAPTER 1

PIP

```
$ pip install enlighten
```

CHAPTER 2

RPM

2.1 Fedora and EL8 (RHEL/CentOS)

(For EPEL repositories must be configured)

```
$ dnf install python3-enlighten
```

CHAPTER 3

DEB

3.1 Debian and Ubuntu

```
$ apt-get install python3-enlighten
```

CHAPTER 4

Conda

```
$ conda install -c conda-forge enlighten
```

CHAPTER 5

How to Use

5.1 Managers

The first step is to create a manager. Managers handle output to the terminal and allow multiple progress bars to be displayed at the same time.

get_manager() can be used to get a *Manager* instance. Managers will only display output when the output stream, `sys.__stdout__` by default, is attached to a TTY. If the stream is not attached to a TTY, the manager instance returned will be disabled.

In most cases, a manager can be created like this.

```
import enlighten
manager = enlighten.get_manager()
```

If you need to use a different output stream, or override the defaults, see the documentation for *get_manager()*

5.2 Progress Bars

For a basic progress bar, invoke the *Manager.counter* method.

```
import time
import enlighten

manager = enlighten.get_manager()
pbar = enlighten.counter(total=100, desc='Basic', unit='ticks')

for num in range(100):
    time.sleep(0.1)  # Simulate work
    pbar.update()
```

Additional progress bars can be created with additional calls to the *Manager.counter* method.

```
import time
import enlighten

manager = enlighten.get_manager()
ticks = manager.counter(total=100, desc='Ticks', unit='ticks')
tocks = manager.counter(total=20, desc='Tocks', unit='tocks')

for num in range(100):
    time.sleep(0.1)  # Simulate work
    print(num)
    ticks.update()
    if not num % 5:
        tocks.update()

manager.stop()
```

5.3 Counters

The *Counter* class has two output formats, progress bar and counter.

The progress bar format is used when a total is not None and the count is less than the total. If neither of these conditions are met, the counter format is used:

```
import time
import enlighten

manager = enlighten.get_manager()
counter = manager.counter(desc='Basic', unit='ticks')

for num in range(100):
    time.sleep(0.1)  # Simulate work
    counter.update()
```

5.4 Status Bars

Status bars are bars that work similarly to progress bars and counters, but present relatively static information. Status bars are created with *Manager.status_bar*.

```
import enlighten
import time

manager = enlighten.get_manager()
status_bar = manager.status_bar('Static Message',
                                color='white_on_red',
                                justify=enlighten.Justify.CENTER)
time.sleep(1)
status_bar.update('Updated static message')
time.sleep(1)
```

Status bars can also use formatting with dynamic variables.

```
import enlighten
import time

manager = enlighten.get_manager()
status_format = '{program}{fill}Stage: {stage}{fill} Status {status}'
status_bar = manager.status_bar(status_format=status_format,
                                color='bold_slategray',
                                program='Demo',
                                stage='Loading',
                                status='OKAY')
time.sleep(1)
status_bar.update(stage='Initializing', status='OKAY')
time.sleep(1)
status_bar.update(status='FAIL')
```

Status bars, like other bars can be pinned. To pin a status bar to the top of all other bars, initialize it before any other bars. To pin a bar to the bottom of the screen, use `position=1` when initializing.

See *StatusBar* for more details.

5.5 Color

Status bars and the bar component of a progress bar can be colored by setting the `color` keyword argument. See *Series Color* for more information about valid colors.

```
import time
import enlighten

manager = enlighten.get_manager()
counter = manager.counter(total=100, desc='Colorized', unit='ticks', color='red')

for num in range(100):
    time.sleep(0.1)  # Simulate work
counter.update()
```

Additionally, any part of the progress bar can be colored using counter *formatting* and the color capabilities of the underlying Blessed Terminal.

```
import enlighten

manager = enlighten.get_manager()

# Standard bar format
std_bar_format = u'{desc}{desc_pad}{percentage:3.0f}%|{bar}| ' + \
                 u'{count:{len_total}d}/{total:d} ' + \
                 u'[{elapsed}<{eta}, {rate:.2f}{unit_pad}{unit}/s]'

# Red text
bar_format = manager.term.red(std_bar_format)

# Red on white background
bar_format = manager.term.red_on_white(std_bar_format)

# X11 colors
bar_format = manager.term.peru_on_seagreen(std_bar_format)
```

(continues on next page)

(continued from previous page)

```
# RBG text
bar_format = manager.term.color_rgb(2, 5, 128)(std_bar_format)

# RBG background
bar_format = manager.term.on_color_rgb(255, 190, 195)(std_bar_format)

# RGB text and background
bar_format = manager.term.on_color_rgb(255, 190, 195)(std_bar_format)
bar_format = manager.term.color_rgb(2, 5, 128)(bar_format)

# Apply color to select parts
bar_format = manager.term.red(u'{desc}') + u'{desc_pad}' + \
             manager.term.blue(u'{percentage:3.0f}%') + u'|{bar}|'

# Apply to counter
ticks = manager.counter(total=100, desc='Ticks', unit='ticks', bar_format=bar_format)
```

If the `color` option is applied to a *`Counter`*, it will override any foreground color applied.

5.6 Multicolored

The bar component of a progress bar can be multicolored to track multiple categories in a single progress bar.

The colors are drawn from right to left in the order they were added.

By default, when multicolored progress bars are used, additional fields are available for `bar_format`:

- count_n (`int`) - Current value of `count`
- count_0(`int`) - Remaining count after deducting counts for all subcounters
- count_00 (`int`) - Sum of counts from all subcounters
- percentage_n (`float`) - Percentage complete
- percentage_0(`float`) - Remaining percentage after deducting percentages for all subcounters
- percentage_00 (`float`) - Total of percentages from all subcounters

When `add_subcounter()` is called with `all_fields` set to `True`, the subcounter will have the additional fields:

- eta_n (`str`) - Estimated time to completion
- rate_n (`float`) - Average increments per second since parent was created

More information about `bar_format` can be found in the *Format* section of the API.

One use case for multicolored progress bars is recording the status of a series of tests. In this example, Failures are red, errors are white, and successes are green. The count of each is listed in the progress bar.

```
import random
import time
import enlighten

bar_format = u'{desc}{desc_pad}{percentage:3.0f}%|{bar}| ' + \
             u'S:{count_0:{len_total}d} ' + \
             u'F:{count_2:{len_total}d} ' + \
```

(continues on next page)

(continued from previous page)

```
            u'E:{count_1:{len_total}d} ' + \
            u'[{elapsed}<{eta}, {rate:.2f}{unit_pad}{unit}/s]'

manager = enlighten.get_manager()
success = manager.counter(total=100, desc='Testing', unit='tests',
                          color='green', bar_format=bar_format)
errors = success.add_subcounter('white')
failures = success.add_subcounter('red')

while success.count < 100:
    time.sleep(random.uniform(0.1, 0.3))  # Random processing time
    result = random.randint(0, 10)

    if result == 7:
        errors.update()
    if result in (5, 6):
        failures.update()
    else:
        success.update()
```

A more complicated example is recording process start-up. In this case, all items will start red, transition to yellow, and eventually all will be green. The count, percentage, rate, and eta fields are all derived from the second subcounter added.

```
import random
import time
import enlighten

services = 100
bar_format = u'{desc}{desc_pad}{percentage_2:3.0f}%|{bar}|' + \
             u' {count_2:{len_total}d}/{total:d} ' + \
             u'[{elapsed}<{eta_2}, {rate_2:.2f}{unit_pad}{unit}/s]'

manager = enlighten.get_manager()
initializing = manager.counter(total=services, desc='Starting', unit='services',
                               color='red', bar_format=bar_format)
starting = initializing.add_subcounter('yellow')
started = initializing.add_subcounter('green', all_fields=True)

while started.count < services:
    remaining = services - initializing.count
    if remaining:
        num = random.randint(0, min(4, remaining))
        initializing.update(num)

    ready = initializing.count - initializing.subcount
    if ready:
        num = random.randint(0, min(3, ready))
        starting.update_from(initializing, num)

    if starting.count:
        num = random.randint(0, min(2, starting.count))
        started.update_from(starting, num)

    time.sleep(random.uniform(0.1, 0.5))  # Random processing time
```

5.7 Additional Examples

- `Basic` - Basic progress bar
- `Binary prefixes` - Automatic binary prefixes
- `Context manager` - Managers and counters as context managers
- `FTP downloader` - Show progress downloading files from FTP
- `Floats` - Support totals and counts that are `floats`
- `Multicolored` - Multicolored progress bars
- `Multiple with logging` - Nested progress bars and logging
- `Multiprocessing queues` - Progress bars with queues for IPC

5.8 Customization

Enlighten is highly configurable. For information on modifying the output, see the *Series* and *Format* sections of the *`Counter`* documentation.

CHAPTER 6

Common Patterns

6.1 Enable / Disable

A program may want to disable progress bars based on a configuration setting as well as if output redirection occurs.

```
import sys
import enlighten

# Example configuration object
config = {'stream': sys.stdout,
          'useCounter': False}

enableCounter = config['useCounter'] and stream.isatty()
manager = enlighten.Manager(stream=config['stream'], enabled=enableCounter)
```

The *get_manager()* function slightly simplifies this

```
import enlighten

# Example configuration object
config = {'stream': None,  # Defaults to sys.__stdout__
          'useCounter': False}

manager = enlighten.get_manager(stream=config['stream'], enabled=config['useCounter'])
```

6.2 Context Managers

Both *Counter* and *Manager* can be used as context managers.

```
import enlighten
```

(continues on next page)

(continued from previous page)

```
SPLINES = 100

with enlighten.Manager() as manager:
    with manager.counter(total=SPLINES, desc='Reticulating:', unit='splines') as␣
↪retic:
        for num in range(SPLINES + 1):
            retic.update()
```

6.3 Automatic Updating

Both *Counter* and *SubCounter* instances can be called as functions on one or more iterators. A generator is returned which yields each element of the iterables and then updates the count by 1.

Note: When a *Counter* instance is called as a function, type checking is lazy and won't validate an iterable was passed until iteration begins.

```
import time
import enlighten

flock1 = ['Harry', 'Sally', 'Randy', 'Mandy', 'Danny', 'Joe']
flock2 = ['Punchy', 'Kicky', 'Spotty', 'Touchy', 'Brenda']
total = len(flock1) + len(flock2)

manager = enlighten.Manager()
pbar = manager.counter(total=total, desc='Counting Sheep', unit='sheep')

for sheep in pbar(flock1, flock2):
    time.sleep(0.2)
    print('%s: Baaa' % sheep)
```

6.4 User-defined fields

Both *Counter* and *StatusBar* accept user defined fields as keyword arguments at initialization and during an update. These fields are persistent and only need to be specified when they change.

In the following example, `source` is a user-defined field that is periodically updated.

```
import enlighten
import random
import time

bar_format = u'{desc}{desc_pad}{source} {percentage:3.0f}%|{bar}| ' + \
             u'{count:{len_total}d}/{total:d} ' + \
             u'[{elapsed}<{eta}, {rate:.2f}{unit_pad}{unit}/s]'
manager = enlighten.get_manager(bar_format=bar_format)

bar = manager.counter(total=100, desc='Loading', unit='files', source='server.a')
for num in range(100):
    time.sleep(0.1)  # Simulate work
    if not num % 5:
```

(continues on next page)

(continued from previous page)

```
        bar.update(source=random.choice(['server.a', 'server.b', 'server.c']))
    else:
        bar.update()
```

For more information, see the *Counter Format* and *StatusBar Format* sections.

6.5 Human-readable numeric prefixes

Enlighten supports automatic SI (metric) and IEC (binary) prefixes using the Prefixed library.

All `rate` and `interval` formatting fields are of the type `prefixed.Float`. `total` and all `count` fields default to `int`. If `total` or or `count` are set to a `float`, or a `float` is provided to `update()`, these fields will be `prefixed.Float` instead.

```
import time
import random
import enlighten

size = random.uniform(1.0, 10.0) * 2 ** 20  # 1-10 MiB (float)
chunk_size = 64 * 1024  # 64 KiB

bar_format = '{desc}{desc_pad}{percentage:3.0f}%|{bar}| ' \
             '{count:!.2j}{unit} / {total:!.2j}{unit} ' \
             '[{elapsed}<{eta}, {rate:!.2j}{unit}/s]'

manager = enlighten.get_manager()
pbar = manager.counter(total=size, desc='Downloading', unit='B', bar_format=bar_
↪format)

bytes_left = size
while bytes_left:
    time.sleep(random.uniform(0.05, 0.15))
    next_chunk = min(chunk_size, bytes_left)
    pbar.update(next_chunk)
    bytes_left -= next_chunk
```

```
import enlighten

counter_format = 'Trying to get to sleep: {count:.2h} sheep'

manager = enlighten.get_manager()
counter = manager.counter(counter_format=counter_format)
counter.count = 0.0
for num in range(10000000):
    counter.update()
```

For more information, see the *Counter Format* and the Prefixed documentation.

6.6 Manually Printing

By default, if the manager's stream is connected to a TTY, bars and counters are automatically printed and updated. There may, however be cases where manual output is desired in addition to or instead of the automatic output. For example, to send to other streams or print to a file.

The output for an individual bar can be retrieved from the `format()` method. This supports optional arguments to specify width and elapsed time.

```
import enlighten

manager = enlighten.get_manager(enabled=False)
pbar = manager.counter(desc='Progress', total=10)

pbar.update()
print(pbar.format(width=100))
```

As a shortcut, the counter object will call the `format()` method with the default arguments when coerced to a string.

```
import enlighten

manager = enlighten.get_manager(enabled=False)
pbar = manager.counter(desc='Progress', total=10)

pbar.update()
print(pbar)
```

While Enlighten's default output provides more advanced capability, A basic refreshing progress bar can be created like so.

```
import enlighten
import time

manager = enlighten.get_manager(enabled=False)
pbar = manager.counter(desc='Progress', total=10)

print()

for num in range(10):
    time.sleep(0.2)
    pbar.update()
    print(f'\r{pbar}', end='', flush=True)

print()
```

CHAPTER 7

Frequently Asked Questions

7.1 Why is Enlighten called Enlighten?

A progress bar's purpose is to inform the user about an ongoing process. Enlighten, meaning "to inform", seems a fitting name. (Plus any names related to progress were already taken)

7.2 Is Windows supported?

Enlighten has supported Windows since version 1.3.0.

Windows does not currently support resizing.

Enlighten also works in Linux-like subsystems for Windows such as Cygwin or Windows Subsystem for Linux.

7.3 Is Jupyter Notebooks Supported?

Support for Jupyter notebooks was added in version 1.10.0.

Jupyter Notebook support is provide by the *NotebookManager* class. If running inside a Jupyter Notebook, *get_manager()* will return a *NotebookManager* instance.

There is currently no support for detecting the width of a Jupyter notebook so output width has been set statically to 100 characters. This can be overridden by passing the `width` keyword argument to *get_manager()*.

7.4 Is PyCharm supported?

PyCharm uses multiple consoles and the behavior differs depending on how the code is called.

Enlighten works natively in the PyCharm command terminal.

To use Enlighten with Run or Debug, terminal emulation must be enabled. Navigate to *Run -> Edit Configurations -> Templates -> Python* and select *Emulate terminal in output console*.

The PyCharm Python console is currently not supported because `sys.stdout` does not reference a valid TTY.

We are also tracking an issue with CSR in the PyCharm terminal.

7.5 Can you add support for _____ terminal?

We are happy to add support for as many terminals as we can. However, not all terminals can be supported. There a few requirements.

1. The terminal must be detectable programmatically

   We need to be able to identify the terminal in some reasonable way and differentiate it from other terminals. This could be through environment variables, the `platform` module, or some other method.

2. A subset of terminal codes must be supported

   While these codes may vary among terminals, the capability must be provided and activated by printing a terminal sequence. The required codes are listed below.

   - move / CUP - Cursor Position
   - hide_cursor / DECTCEM - Text Cursor Enable Mode
   - show_cursor / DECTCEM - Text Cursor Enable Mode
   - csr / DECSTBM - Set Top and Bottom Margins
   - clear_eos / ED - Erase in Display
   - clear_eol / EL - Erase in Line
   - feed / CUD - Cursor Down (Or scroll with linefeed)

3. Terminal dimensions must be detectable

   The height and width of the terminal must be available to the running process.

7.6 Why does `RuntimeError: reentrant call` get raised sometimes during a resize?

This is caused when another thread or process is writing to a standard stream (STDOUT, STDERR) at the same time the resize signal handler is writing to the stream.

Enlighten tries to detect when a program is threaded or running multiple processes and defer resize handling until the next normal write event. However, this condition is evaluated when the scroll area is set, typically when the first counter is added. If no threads or processes are detected at that time, and the value of threaded was not set explicitly, resize events will not be deferred.

In order to guarantee resize handling is deferred, it is best to pass `threaded=True` when creating a manager instance.

7.7 Why isn't my progress bar displayed until `update()` is called?

Progress bars and counters are not automatically drawn when created because some fields may be missing if subcounters are used. To force the counter to draw before updating, call *refresh()*

CHAPTER 8

API Reference

8.1 Classes

class enlighten.**Manager**(*stream=None*, *counter_class=Counter*, ***kwargs*)

Parameters

- **stream** (file object) – Output stream. If `None`, defaults to `sys.__stdout__`
- **status_bar_class** (class) – Status bar class (Default: *`StatusBar`*)
- **counter_class** (class) – Progress bar class (Default: *`Counter`*)
- **set_scroll** (*`bool`*) – Enable scroll area redefinition (Default: `True`)
- **companion_stream** (file object) – See *companion_stream* below. (Default: `None`)
- **enabled** (*`bool`*) – Status (Default: True)
- **no_resize** (*`bool`*) – Disable resizing support
- **threaded** (*`bool`*) – When True resize handling is deferred until next write (Default: False unless multiple threads or multiple processes are detected)
- **width** (*`int`*) – Static output width. If unset, terminal width is determined dynamically
- **kwargs** (*`Dict[str, Any]`*) – Any additional keyword arguments will be used as default values when *`counter()`* is called.

Manager class for outputting progress bars to streams attached to TTYs

Progress bars are displayed at the bottom of the screen with standard output displayed above.

companion_stream

A companion stream is a file object that shares a TTY with the primary output stream. The cursor position in the companion stream will be moved in coordination with the primary stream.

If the value is `None`, the companion stream will be dynamically determined. Unless explicitly specified, a stream which is not attached to a TTY (the case when redirected to a file), will not be used as a companion stream.

counter(*position=None*, ***kwargs*)

Parameters

- **position** (*int*) – Line number counting from the bottom of the screen
- **autorefresh** (*bool*) – Refresh this counter when other bars are drawn
- **replace** (`PrintableCounter`) – Replace given counter with new. Position ignored.
- **kwargs** (*Dict[str, Any]*) – Any additional keyword arguments are passed to *Counter*

Returns Instance of counter class

Return type *Counter*

Get a new progress bar instance

If `position` is specified, the counter's position will be pinned. A `ValueError` will be raised if `position` exceeds the screen height or has already been pinned by another counter.

If `autorefresh` is `True`, this bar will be redrawn whenever another bar is drawn assuming it had been `min_delta` seconds since the last update. This is usually unnecessary.

Note: Counters are not automatically drawn when created because fields may be missing if subcounters are used. To force the counter to draw before updating, call *refresh()*.

---

status_bar(**args*, ***kwargs*)

Parameters

- **position** (*int*) – Line number counting from the bottom of the screen
- **autorefresh** (*bool*) – Refresh this counter when other bars are drawn
- **replace** (`PrintableCounter`) – Replace given counter with new. Position ignored.
- **kwargs** (*Dict[str, Any]*) – Any additional keyword arguments are passed to *StatusBar*

Returns Instance of status bar class

Return type *StatusBar*

Get a new status bar instance

If `position` is specified, the counter's position can change dynamically if additional counters are called without a `position` argument.

If `autorefresh` is `True`, this bar will be redrawn whenever another bar is drawn assuming it had been `min_delta` seconds since the last update. Generally, only need when `elapsed` is used in *status_format*.

stop()

Clean up and reset terminal

This method should be called when the manager and counters will no longer be needed.

Any progress bars that have `leave` set to `True` or have not been closed will remain on the console. All others will be cleared.

Manager and all counters will be disabled.

class enlighten.**NotebookManager**(*stream=None*, *counter_class=Counter*, ***kwargs*)

Parameters

- **counter_class** (class) – Progress bar class (Default: *Counter*)
- **status_bar_class** (class) – Status bar class (Default: *StatusBar*)
- **enabled** (*bool*) – Status (Default: True)
- **width** (*int*) – Static output width (Default: 100)
- **kwargs** (*Dict[str, Any]*) – Any additional keyword arguments will be used as default values when *counter()* is called.

Manager class for outputting progress bars to Jupyter notebooks

The following keyword arguments are set if provided, but ignored:

- *stream*
- *set_scroll*
- *companion_stream*
- *no_resize*
- *threaded*

counter(*position=None*, ***kwargs*)

Parameters

- **position** (*int*) – Line number counting from the bottom of the screen
- **autorefresh** (*bool*) – Refresh this counter when other bars are drawn
- **replace** (PrintableCounter) – Replace given counter with new. Position ignored.
- **kwargs** (*Dict[str, Any]*) – Any additional keyword arguments are passed to *Counter*

Returns Instance of counter class

Return type *Counter*

Get a new progress bar instance

If `position` is specified, the counter's position will be pinned. A `ValueError` will be raised if `position` exceeds the screen height or has already been pinned by another counter.

If `autorefresh` is `True`, this bar will be redrawn whenever another bar is drawn assuming it had been `min_delta` seconds since the last update. This is usually unnecessary.

Note: Counters are not automatically drawn when created because fields may be missing if subcounters are used. To force the counter to draw before updating, call *refresh()*.

---

status_bar(**args*, ***kwargs*)

Parameters

- **position** (*int*) – Line number counting from the bottom of the screen
- **autorefresh** (*bool*) – Refresh this counter when other bars are drawn

- **replace** (PrintableCounter) – Replace given counter with new. Position ignored.
- **kwargs** (*Dict[str, Any]*) – Any additional keyword arguments are passed to *StatusBar*

Returns Instance of status bar class

Return type *StatusBar*

Get a new status bar instance

If `position` is specified, the counter's position can change dynamically if additional counters are called without a `position` argument.

If `autorefresh` is `True`, this bar will be redrawn whenever another bar is drawn assuming it had been `min_delta` seconds since the last update. Generally, only need when `elapsed` is used in *status_format*.

stop()

Clean up and reset terminal

This method should be called when the manager and counters will no longer be needed.

Any progress bars that have `leave` set to `True` or have not been closed will remain on the console. All others will be cleared.

Manager and all counters will be disabled.

class enlighten.**Counter**(***kwargs*)

Parameters

- **all_fields** (*bool*) – Populate `rate`, `interval`, and `eta` formatting fields in sub-counters
- **bar_format** (*str*) – Progress bar format, see *Format* below
- **count** (*int*) – Initial count (Default: 0)
- **counter_format** (*str*) – Counter format, see *Format* below
- **color** (*str*) – Series color as a string or RGB tuple see *Series Color*
- **desc** (*str*) – Description
- **enabled** (*bool*) – Status (Default: `True`)
- **fill** (*str*) – Fill character used for `counter_format` (Default: ' ')
- **fields** (*dict*) – Additional fields used for *formatting*
- **leave** (*True*) – Leave progress bar after closing (Default: `True`)
- **manager** (*Manager*) – Manager instance. Creates instance if not specified.
- **min_delta** (*float*) – Minimum time, in seconds, between refreshes (Default: 0.1)
- **offset** (*int*) – Number of non-printable characters to account for when formatting
- **series** (sequence) – Progression series, see *Series* below
- **stream** (file object) – Output stream. Not used when instantiated through a manager
- **total** (*int*) – Total count when complete
- **unit** (*str*) – Unit label

Progress bar and counter class

A *Counter* instance can be created with the *Manager.counter()* method or, when a standalone progress bar for simple applications is required, the *Counter* class can be called directly. The output stream will default to sys.__stdout__ unless stream is set.

Note: With the default values for bar_format and counter_format, floats can not be used for total, count, or provided to *update()*. In order to use floats, provide custom formats to bar_format and counter_format. See *Format* below.

Series

The progress bar is constructed from the characters in series. series must be a sequence (str, list, tuple) containing single characters.

Default progress series (series):

```
' '
```

The first character is the fill character. When the count is 0, the bar will be made up of only this character. In the example below, characters 5 through 9 are fill characters.

The last character is the full character. When the count is equal to total, the bar will be made up of only this character. In the example below, characters 0 through 3 are full characters.

The remaining characters are fractional characters used to more accurately represent the transition between the full and fill characters. In the example below, character 4 is a fractional character.

```
'45% |     |'
     '0123456789'
```

Series Color

The characters specified by series will be displayed in the terminal's current foreground color. This can be overwritten with the color argument.

color can be specified as None, a string or, an iterable of three integers, 0 - 255, describing an RGB color.

For backward compatibility, a color can be expressed as an integer 0 - 255, but this is deprecated in favor of named or RGB colors.

Compound colors, such as 'white_on_seagreen', 'bold_red', or 'underline_on_peru' are also supported.

If a terminal is not capable of 24-bit color, and is given a color outside of its range, the color will be downconverted to a supported color.

Valid colors for 8 color terminals:

- black
- blue
- cyan
- green
- magenta
- red
- white

- yellow

Additional colors for 16 color terminals:

- bright_black
- bright_blue
- bright_cyan
- bright_green
- bright_magenta
- bright_red
- bright_white
- bright_yellow

See this chart for a complete list of supported color strings.

Note: If an invalid color is specified, an `AttributeError` will be raised

---

Format

If `total` is `None` or `count` becomes higher than `total`, the counter format will be used instead of the progress bar format.

Default counter format (`counter_format`):

```
'{desc}{desc_pad}{count:d} {unit}{unit_pad}{elapsed}, {rate:.2f}{unit_pad}
↪{unit}/s]{fill}'

# Example output
'Loaded 30042 Files [00:01, 21446.45 Files/s]
↪       '
```

Default progress bar format (`bar_format`):

```
'{desc}{desc_pad}{percentage:3.0f}%|{bar}| {count:{len_total}d}/{total:d}
↪[{elapsed}<{eta}, {rate:.2f}{unit_pad}{unit}/s]'

# Example output
'Processing    22%|                   |  23/101 [00:27<01:32, 0.84 Files/
↪s]'
```

Available fields:

- count(`int`) - Current value of `count`
- desc(`str`) - Value of `desc`
- desc_pad(`str`) - A single space if `desc` is set, otherwise empty
- elapsed(`str`) - Time elapsed since instance was created
- interval(`prefixed.Float`) - Average seconds per iteration (inverse of rate)
- rate(`prefixed.Float`) - Average iterations per second since instance was created
- total(`int`) - Value of `total`
- unit(`str`) - Value of `unit`

- unit_pad(`str`) - A single space if `unit` is set, otherwise empty

Additional fields for `bar_format` only:

- bar(`str`) - Progress bar draw with characters from `series`
- eta(`str`) - Estimated time to completion
- len_total(`int`) - Length of `total` when converted to a string
- percentage(`float`) - Percentage complete

Additional fields for `counter_format` only:

- fill(`str`) - Filled with *`fill`* until line is width of terminal. May be used multiple times. Minimum width is 3.

Additional fields when subcounters are used:

- count_n(`int`) - Current value of `count`
- count_0(`int`) - Remaining count after deducting counts for all subcounters
- count_00(`int`) - Sum of counts from all subcounters
- interval_0(`prefixed.Float`) - Average seconds per non-subcounter iteration (inverse of rate_0)
- interval_00(`prefixed.Float`) - Average seconds per iteration for all subcounters (inverse of rate_00)
- percentage_n(`float`) - Percentage complete (`bar_format` only)
- percentage_0(`float`) - Remaining percentage after deducting percentages for all subcounters (`bar_format` only)
- percentage_00(`float`) - Total of percentages from all subcounters
- rate_0(`prefixed.Float`) - Average iterations per second excluding subcounters since instance was created
- rate_00(`prefixed.Float`) - Average iterations per second of all subcounters since instance was created

---

Note: **n** denotes the order the subcounter was added starting at 1. For example, **count_1** is the count for the first subcounter added and **count_2** is the count for the second subcounter added.

Additional fields when *`add_subcounter()`* is called with `all_fields` set to `True`:

- eta_n (`str`) - Estimated time to completion (`bar_format` only)
- interval_n(`prefixed.Float`) - Average seconds per iteration (inverse of rate)
- rate_n (`prefixed.Float`) - Average iterations per second since parent was created

---

Note: `count` and `total` fields, including `count_0`, `count_00`, and `count_n`, default to `int`. If *`total`* or or *`count`* are set to a `float`, or a `float` is provided to *`update()`*, these fields will be `prefixed.Float` instead.

This allows additional format specifiers using SI (metric) and IEC (binary) prefixes. See the Prefixed documentation for more details.

This will also require a custom format and affect the accuracy of the `len_total` field.

User-defined fields:

Users can define fields in two ways, the `fields` parameter and by passing keyword arguments to *`Manager.counter()`* or *`Counter.update()`*

The `fields` parameter can be used to pass a dictionary of additional user-defined fields. The dictionary values can be updated after initialization to allow for dynamic fields. Any fields that share names with built-in fields are ignored.

If fields are passed as keyword arguments to *`Manager.counter()`* or *`Counter.update()`*, they take precedent over the `fields` parameter.

Offset

When `offset` is `None`, the width of the bar portion of the progress bar and the fill size for counter will be automatically determined, taking into account terminal escape sequences that may be included in the string.

Under special circumstances, and to permit backward compatibility, `offset` may be explicitly set to an `int` value. When explicitly set, automatic detection of escape sequences is disabled.

Instance Attributes

`count`
`int` - Current count

`desc`
`str` - Description

`elapsed`
`float` - Time since start (since last update if `count`equals :py:attr:`total`)

`enabled`
`bool` - Current status

`manager`
`Manager` - Manager Instance

`position`
`int` - Current position

`total`
`int` - Total count when complete

`unit`
`str` - Unit label

`add_subcounter`(*color*, *count=0*, *all_fields=None*)

Parameters

- **`color`** (*`str`*) – Series color as a string or RGB tuple see *Series Color*
- **`count`** (*`int`*) – Initial count (Default: 0)
- **`all_fields`** (*`bool`*) – Populate `rate`, `interval`, and `eta` formatting fields (Default: False unless specified in parent)

Returns Subcounter instance

Return type *`SubCounter`*

Add a subcounter for multicolored progress bars

`clear`(*flush=True*)

Parameters **flush** (*bool*) – Flush stream after clearing bar (Default:True)

Clear bar

close(*clear=False*)
Do final refresh and remove from manager

If `leave` is True, the default, the effect is the same as *refresh()*.

color
Color property

Preferred to be a string or iterable of three integers for RGB. Single integer supported for backwards compatibility

fill
Fill character used in formatting

format(*width=None*, *elapsed=None*)

Parameters

- **width** (*int*) – Width in columns to make progress bar
- **elapsed** (*float*) – Time since started. Automatically determined if `None`

Returns Formatted progress bar or counter

Return type `str`

Format progress bar or counter

refresh(*flush=True*, *elapsed=None*)

Parameters

- **flush** (*bool*) – Flush stream after writing bar (Default:True)
- **elapsed** (*float*) – Time since started. Automatically determined if `None`

Redraw bar

subcount
Sum of counts from all subcounters

update(*incr=1*, *force=False*, ***fields*)

Parameters

- **incr** (*int*) – Amount to increment `count` (Default: 1)
- **force** (*bool*) – Force refresh even if `min_delta` has not been reached
- **fields** (*dict*) – Fields for for *formatting*

Increment progress bar and redraw

Progress bar is only redrawn if `min_delta` seconds past since the last update

class enlighten.**StatusBar**(**args*, ***kwargs*)

Parameters

- **enabled** (*bool*) – Status (Default: `True`)
- **color** (*str*) – Color as a string or RGB tuple see *Status Color*
- **fields** (*dict*) – Additional fields used for *formating*
- **fill** (*str*) – Fill character used in formatting and justifying text (Default: ' ')

- **justify** (*str*) – One of *Justify.CENTER*, *Justify.LEFT*, *Justify.RIGHT*
- **leave** (*True*) – Leave status bar after closing (Default: True)
- **min_delta** (*float*) – Minimum time, in seconds, between refreshes (Default: 0.1)
- **status_format** (*str*) – Status bar format, see *Format*

Status bar class

A *StatusBar* instance should be created with the *Manager.status_bar()* method.

Status Color

Color works similarly to color on *Counter*, except it affects the entire status bar. See *Series Color* for more information.

Format

There are two ways to populate the status bar, direct and formatted. Direct takes precedence over formatted.

Direct Status

Direct status is used when arguments are passed to *Manager.status_bar()* or *StatusBar.update()*. Any arguments are coerced to strings and joined with a space. For example:

```
status_bar.update('Hello', 'World!')
# Example output: Hello World!

status_bar.update('Hello World!')
# Example output: Hello World!

count = [1, 2, 3, 4]
status_bar.update(*count)
 # Example output: 1 2 3 4
```

Formatted Status

Formatted status uses the format specified in the `status_format` parameter to populate the status bar.

```
'Current Stage: {stage}'

# Example output
'Current Stage: Testing'
```

Available fields:

- elapsed(`str`) - Time elapsed since instance was created
- fill(`str`) - Filled with *fill* until line is width of terminal. May be used multiple times. Minimum width is 3.

Note: The status bar is only updated when *StatusBar.update()* or *StatusBar.refresh()* is called, so fields like `elapsed` will need additional calls to appear dynamic.

User-defined fields:

Users can define fields in two ways, the `fields` parameter and by passing keyword arguments to *Manager.status_bar()* or *StatusBar.update()*

The `fields` parameter can be used to pass a dictionary of additional user-defined fields. The dictionary values can be updated after initialization to allow for dynamic fields. Any fields that share names with available fields are ignored.

If fields are passed as keyword arguments to *`Manager.status_bar()`* or *`StatusBar.update()`*, they take precedent over the `fields` parameter.

Instance Attributes

elapsed
`float` - Time since start

enabled
`bool` - Current status

manager
`Manager` - Manager Instance

position
`int` - Current position

clear(*flush=True*)

Parameters **flush** (*bool*) – Flush stream after clearing bar (Default:True)

Clear bar

close(*clear=False*)
Do final refresh and remove from manager

If `leave` is True, the default, the effect is the same as *`refresh()`*.

color
Color property

Preferred to be a string or iterable of three integers for RGB. Single integer supported for backwards compatibility

fill
Fill character used in formatting

format(*width=None*, *elapsed=None*)

Parameters

- **width** (*int*) – Width in columns to make progress bar
- **elapsed** (*float*) – Time since started. Automatically determined if `None`

Returns Formatted status bar

Return type `str`

Format status bar

justify
Maps to justify method determined by `justify` parameter

refresh(*flush=True*, *elapsed=None*)

Parameters

- **flush** (*bool*) – Flush stream after writing bar (Default:True)
- **elapsed** (*float*) – Time since started. Automatically determined if `None`

Redraw bar

update(**objects*, ***fields*)

Parameters

- **objects** (*list*) – Values for *Direct Status*
- **force** (*bool*) – Force refresh even if `min_delta` has not been reached
- **fields** (*dict*) – Fields for for *Formatted Status*

Update status and redraw

Status bar is only redrawn if `min_delta` seconds past since the last update

class enlighten.**SubCounter**(*parent*, *color=None*, *count=0*, *all_fields=False*)

A child counter for multicolored progress bars.

This class tracks a portion of multicolored progress bar and should be initialized through *Counter.add_subcounter()*

Instance Attributes

count
int - Current count

parent
Counter - Parent counter

update(*incr=1*, *force=False*)

Parameters

- **incr** (*int*) – Amount to increment `count` (Default: 1)
- **force** (*bool*) – Force refresh even if `min_delta` has not been reached

Increment progress bar and redraw

Both this counter and the parent are incremented.

Progress bar is only redrawn if min_delta seconds past since the last update on the parent.

update_from(*source*, *incr=1*, *force=False*)

Parameters

- **source** (*SubCounter*) – *SubCounter* or *Counter* to increment from
- **incr** (*int*) – Amount to increment `count` (Default: 1)
- **force** (*bool*) – Force refresh even if `min_delta` has not been reached

Move a value to this counter from another counter.

`source` must be the parent *Counter* instance or a *SubCounter* with the same parent

8.2 Functions

enlighten.**get_manager**(*stream=None*, *counter_class=Counter*, ***kwargs*)

Parameters

- **stream** (file object) – Output stream. If `None`, defaults to `sys.__stdout__`
- **counter_class** (class) – Progress bar class (Default: *Counter*)

- **kwargs** (*Dict[str, Any]*) – Any additional keyword arguments will passed to the manager class.

Returns Manager instance

Return type *Manager*

Convenience function to get a manager instance

If running inside a notebook, a *NotebookManager* instance is returned. otherwise a standard *Manager* instance is returned.

If a a standard *Manager* instance is used and `stream` is not attached to a TTY, the *Manager* instance is disabled.

8.3 Constants

class enlighten.**Justify**

Enumerated type for justification options

CENTER

Justify center

LEFT

Justify left

RIGHT

Justify right

CHAPTER 9

Overview

Enlighten Progress Bar is a console progress bar library for Python.

The main advantage of Enlighten is it allows writing to stdout and stderr without any redirection or additional code. Just print or log as you normally would.

Enlighten also includes experimental support for Jupyter Notebooks.

The code for this animation can be found in demo.py in examples.

Python Module Index

e

enlighten, 23

Index

A

add_subcounter() (*enlighten.Counter method*), 30

C

CENTER (*enlighten.Justify attribute*), 35
clear() (*enlighten.Counter method*), 30
clear() (*enlighten.StatusBar method*), 33
close() (*enlighten.Counter method*), 31
close() (*enlighten.StatusBar method*), 33
color (*enlighten.Counter attribute*), 31
color (*enlighten.StatusBar attribute*), 33
count (*enlighten.Counter attribute*), 30
count (*enlighten.SubCounter attribute*), 34
Counter (*class in enlighten*), 26
counter() (*enlighten.Manager method*), 24
counter() (*enlighten.NotebookManager method*), 25

D

desc (*enlighten.Counter attribute*), 30

E

elapsed (*enlighten.Counter attribute*), 30
elapsed (*enlighten.StatusBar attribute*), 33
enabled (*enlighten.Counter attribute*), 30
enabled (*enlighten.StatusBar attribute*), 33
enlighten (*module*), 23

F

fill (*enlighten.Counter attribute*), 31
fill (*enlighten.StatusBar attribute*), 33
format() (*enlighten.Counter method*), 31
format() (*enlighten.StatusBar method*), 33

G

get_manager() (*in module enlighten*), 34

J

Justify (*class in enlighten*), 35
justify (*enlighten.StatusBar attribute*), 33

L

LEFT (*enlighten.Justify attribute*), 35

M

Manager (*class in enlighten*), 23
manager (*enlighten.Counter attribute*), 30
manager (*enlighten.StatusBar attribute*), 33

N

NotebookManager (*class in enlighten*), 25

P

parent (*enlighten.SubCounter attribute*), 34
position (*enlighten.Counter attribute*), 30
position (*enlighten.StatusBar attribute*), 33

R

refresh() (*enlighten.Counter method*), 31
refresh() (*enlighten.StatusBar method*), 33
RIGHT (*enlighten.Justify attribute*), 35

S

status_bar() (*enlighten.Manager method*), 24
status_bar() (*enlighten.NotebookManager method*), 25
StatusBar (*class in enlighten*), 31
stop() (*enlighten.Manager method*), 24
stop() (*enlighten.NotebookManager method*), 26
subcount (*enlighten.Counter attribute*), 31
SubCounter (*class in enlighten*), 34

T

total (*enlighten.Counter attribute*), 30

U

unit (*enlighten.Counter attribute*), 30
update() (*enlighten.Counter method*), 31
update() (*enlighten.StatusBar method*), 33
update() (*enlighten.SubCounter method*), 34
update_from() (*enlighten.SubCounter method*), 34