Enlighten Documentation

Release 1.5.1

Avram Lubkin

Mar 29, 2020

Contents

CHAPTER 1

PIP

```
$ pip install enlighten
```

CHAPTER 2

RPM

RPMs are available in the Fedora and EPEL repositories

2.1 EL6 and EL7 (RHEL/CentOS/Scientific)

(EPEL repositories must be configured)

```
$ yum install python-enlighten
```

2.2 Fedora

```
$ dnf install python2-enlighten
$ dnf install python3-enlighten
```

CHAPTER 3

Examples

3.1 Basic

For a basic status bar, invoke the *Counter* class directly.

```
import time
import enlighten

pbar = enlighten.Counter(total=100, desc='Basic', unit='ticks')
for num in range(100):
    time.sleep(0.1)  # Simulate work
    pbar.update()
```

3.2 Advanced

To maintain multiple progress bars simultaneously or write to the console, a manager is required.

Advanced output will only work when the output stream, `sys.stdout` by default, is attached to a TTY. *get_manager()* can be used to get a manager instance. It will return a disabled *Manager* instance if the stream is not attached to a TTY and an enabled instance if it is.

```
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()
```

(continues on next page)

(continued from previous page)

```
    if not num % 5:
        tocks.update()

manager.stop()
```

3.3 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

counter = enlighten.Counter(desc='Basic', unit='ticks')
for num in range(100):
    time.sleep(0.1)  # Simulate work
    counter.update()
```

3.4 Color

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

counter = enlighten.Counter(total=100, desc='Colorized', unit='ticks', color='red')
for num in range(100):
    time.sleep(0.1)  # Simulate work
counter.update()
```

3.5 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
- percentage_n (float) - Percentage complete
- percentage_0(float) - Remaining percentage after deducting percentages for 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} ' + \
             u'E:{count_1:{len_total}d} ' + \
             u'[{elapsed}<{eta}, {rate:.2f}{unit_pad}{unit}/s]'

success = enlighten.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]'

initializing = enlighten.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
```

(continues on next page)

(continued from previous page)

```
    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
```

3.6 Additional Examples

- `basic` - Basic progress bar
- `context manager` - Managers and counters as context managers
- `floats` - Support totals and counts that are `floats`
- `multicolored` - Multicolored progress bars
- `multiple with logging` - Nested progress bars and logging
- `FTP downloader` - Show progress downloading files from FTP
- `Multiprocessing queues` - Progress bars with queues for IPC

3.7 Customization

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

CHAPTER 4

Common Patterns

4.1 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'])
```

4.2 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()
```

4.3 Automatic Updating

Both *Counter* and Both *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)
```

CHAPTER 5

Frequently Asked Questions

5.1 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)

5.2 Is Windows supported?

Enlighten has supported Windows since version 1.3.0.

Windows does not currently support resizing.

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

5.3 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.

5.4 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.

CHAPTER 6

API Reference

6.1 Classes

class enlighten.**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`*)
- **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
- **kwargs** (*`dict`*) – 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`, `sys.stdout` and `sys.stderr` will be used as companion streams. 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

- **kwargs** (*dict*) – 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 can change dynamically if additional counters are called without a `position` argument.

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

- **additional_fields** (*dict*) – Additional fields used for *formating*
- **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`)
- **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.

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
- rate(`float`) - Average increments per second since instance was created
- 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
- total(`int`) - Value of `total`

Addition fields for `counter_format` only:

- fill(`str`) - blank spaces, number needed to fill line

Additional fields when subcounters are used:

- count_n (`int`) - Current value of `count`

- count_0(int) - Remaining count after deducting counts for all subcounters
- percentage_n (float) - Percentage complete (bar_format only)
- percentage_0(float) - Remaining percentage after deducting percentages for all subcounters (bar_format only)

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)
- rate_n (float) - Average increments per second since parent was created

User-defined fields:

The additional_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.

Offset

When offset is None, the width of the bar portion of the progress bar and the fill characters 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=False*)

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` and `eta` formatting fields (Default: False)

Returns Subcounter instance

Return type *SubCounter*

Add a subcounter for multicolored progress bars

clear(*flush=True*)

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

Clear progress 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

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 progress bar (Default:True)
- **elapsed** (*float*) – Time since started. Automatically determined if `None`

Redraw progress bar

subcount

Sum of counts from all subcounters

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

Progress 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

6.2 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*) – Any additional keyword arguments will passed to the manager class.

Returns Manager instance

Return type *Manager*

Convenience function to get a manager instance

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

CHAPTER 7

Overview

Enlighten Progress Bar is a console progress bar module for Python. (Yes, another one.) The main advantage of Enlighten is it allows writing to stdout and stderr without any redirection.

Python Module Index

e

Index