progressive Documentation

Release 0.3.4

Hamza Faran

Dec 18, 2017

Contents

`progressive` allows you to progress of complex (and simple) workflows in your terminal.

If your aim is to draw progress bars, nested or otherwise, you've come to the right place.

```
from progressive.bar import Bar

bar = Bar(max_value=100)
bar.cursor.clear_lines(2)  # Make some room
bar.cursor.save()  # Mark starting line
for i in range(101):
    sleep(0.1)  # Do some work
    bar.cursor.restore()  # Return cursor to start
    bar.draw(value=i)  # Draw the bar!
```

More examples are available in `progressive.examples`!

```
# -*- coding: utf-8 -*-
"""Examples

Usage:
`python -c "from progressive.examples import *; tree()"` as an example
"""
import random
from time import sleep

from blessings import Terminal

from progressive.bar import Bar
from progressive.tree import ProgressTree, Value, BarDescriptor


def simple():
    """Simple example using just the Bar class

    This example is intended to show usage of the Bar class at the lowest
    level.
    """
    MAX_VALUE = 100

    # Create our test progress bar
    bar = Bar(max_value=MAX_VALUE, fallback=True)

    bar.cursor.clear_lines(2)
     # Before beginning to draw our bars, we save the position
    #   of our cursor so we can restore back to this position before writing
    #   the next time.
    bar.cursor.save()
    for i in range(MAX_VALUE + 1):
        sleep(0.1 * random.random())
        # We restore the cursor to saved position before writing
        bar.cursor.restore()
        # Now we draw the bar
        bar.draw(value=i)


def tree():
    """Example showing tree progress view"""
```

```
    #############
    # Test data #
    #############

    # For this example, we're obviously going to be feeding fictitious data
    #   to ProgressTree, so here it is
    leaf_values = [Value(0) for i in range(6)]
    bd_defaults = dict(type=Bar, kwargs=dict(max_value=10))

    test_d = {
        "Warp Jump": {
            "1) Prepare fuel": {
                "Load Tanks": {
                    "Tank 1": BarDescriptor(value=leaf_values[0], **bd_defaults),
                    "Tank 2": BarDescriptor(value=leaf_values[1], **bd_defaults),
                },
                "Refine tylium ore": BarDescriptor(
                    value=leaf_values[2], **bd_defaults
                ),
            },
            "2) Calculate jump co-ordinates": {
                "Resolve common name to co-ordinates": {
                    "Querying resolution from baseship": BarDescriptor(
                        value=leaf_values[3], **bd_defaults
                    ),
                },
            },
            "3) Perform jump": {
                "Check FTL drive readiness": BarDescriptor(
                    value=leaf_values[4], **bd_defaults
                ),
                "Juuuuuump!": BarDescriptor(value=leaf_values[5],
                                            **bd_defaults)
            }
        }
    }

    # We'll use this function to bump up the leaf values
    def incr_value(obj):
        for val in leaf_values:
            if val.value < 10:
                val.value += 1
                break

    # And this to check if we're to stop drawing
    def are_we_done(obj):
        return all(val.value == 10 for val in leaf_values)

    ###################
    # The actual code #
    ###################

    # Create blessings.Terminal instance
    t = Terminal()
    # Initialize a ProgressTree instance
    n = ProgressTree(term=t)
    # We'll use the make_room method to make sure the terminal
```

```
    #   is filled out with all the room we need
    n.make_room(test_d)

    while not are_we_done(test_d):
        sleep(0.2 * random.random())
        # After the cursor position is first saved (in the first draw call)
        #   this will restore the cursor back to the top so we can draw again
        n.cursor.restore()
        # We use our incr_value method to bump the fake numbers
        incr_value(test_d)
        # Actually draw out the bars
        n.draw(test_d, BarDescriptor(bd_defaults))
```

CHAPTER 1

API Reference

If you are looking for information on a specific function, class, or method, this part of the documentation is for you.

1.1 progressive

1.1.1 progressive.bar module

class progressive.bar.**Bar**(*term=None*, *max_value=100*, *width=u'25%'*, *title_pos=u'left'*, *title=u'Progress'*, *num_rep=u'fraction'*, *indent=0*, *filled_color=2*, *empty_color=7*, *back_color=None*, *filled_char=u' '*, *empty_char=u' '*, *start_char=u''*, *end_char=u''*, *fallback=True*, *fallback_empty_char=u'u25ef'*, *fallback_filled_char=u'u25c9'*, *force_color=None*)

Bases: object

Progress Bar with blessings

Several parts of this class are thanks to Erik Rose's implementation of ProgressBar in nose-progressive, licensed under The MIT License. MIT nose-progressive/noseprogressive/bar.py

Terminal with 256 colors is recommended. See this for Ubuntu installation as an example.

Parameters

- **term** (*blessings.Terminal|NoneType*) – blessings.Terminal instance for the terminal of display
- **max_value** (*int*) – The capacity of the bar, i.e., value/max_value
- **width** (*str*) – Must be of format {num: int}{unit: c|%}. Unit "c" can be used to specify number of maximum columns; unit "%". to specify percentage of the total terminal width to use. e.g., "20c", "25%", etc.
- **title_pos** (*str*) – Position of title relative to the progress bar; can be any one of ["left", "right", "above", "below"]

- **title** (*str*) – Title of the progress bar
- **num_rep** (*str*) – Numeric representation of completion; can be one of ["fraction", "percentage"]
- **indent** (*int*) – Spaces to indent the bar from the left-hand side
- **filled_color** (*str|int*) – color of the `filled_char`; can be a string of the color's name or number representing the color; see the `blessings` documentation for details
- **empty_color** (*str|int*) – color of the `empty_char`
- **back_color** (*str|NoneType*) – Background color of the progress bar; must be a string of the color name, unused if numbers used for `filled_color` and `empty_color`. If set to None, will not be used.
- **filled_char** (*unicode*) – Character representing completeness on the progress bar
- **empty_char** (*unicode*) – The complement to `filled_char`
- **start_char** (*unicode*) – Character at the start of the progress bar
- **end_char** (*unicode*) – Character at the end of the progress bar
- **fallback** (*bool*) – If this is set, if the terminal does not support provided colors, this will fall back to plain formatting that works on terminals with no color support, using the provided `fallback_empty_char` and ``fallback_filled_char`
- **force_color** (*bool|NoneType*) – `True` forces color to be used even if it may not be supported by the terminal; `False` forces use of the fallback formatting; `None` does not force anything and allows automatic detection as usual.

draw(*value*, *newline=True*, *flush=True*)
Draw the progress bar

Parameters

- **value** (*int*) – Progress value relative to `self.max_value`
- **newline** (*bool*) – If this is set, a newline will be written after drawing

empty
Callable for drawing empty portion of progress bar

Return type callable

end_char
Character at the end of the progress bar

filled
Callable for drawing filled portion of progress bar

Return type callable

full_line_width
Find actual length of bar_str

e.g., Progress [|] 10/10

max_value
The capacity of the bar, i.e., `value/max_value`

max_width
Get maximum width of progress bar

Return type int

Returns Maximum column width of progress bar

start_char
Character at the start of the progress bar

title
Title of the progress bar

1.1.2 progressive.cursor module

class progressive.cursor.**Cursor**(*term=None*)
Bases: object

Common methods for cursor manipulation

Parameters **term** (*NoneType|blessings.Terminal*) – Terminal instance; if not given, will be created by the class

clear_lines(*num_lines=0*)

flush()
Flush buffer of terminal output stream

newline()
Effects a newline by moving the cursor down and clearing

restore()
Restores cursor to the previously saved location

Cursor position will only be restored IF it was previously saved by this instance (and not by any external force)

save()
Saves current cursor position, so that it can be restored later

write(*s*)
Writes s to the terminal output stream

Writes can be disabled by setting the environment variable *PROGRESSIVE_NOWRITE* to *'True'*

1.1.3 progressive.examples module

Examples

Usage: *python -c "from progressive.examples import *; tree()"* as an example

progressive.examples.**simple**()
Simple example using just the Bar class

This example is intended to show usage of the Bar class at the lowest level.

progressive.examples.**tree**()
Example showing tree progress view

1.1.4 progressive.exceptions module

exception progressive.exceptions.**ColorUnsupportedError**
Bases: *progressive.exceptions.ProgressiveException*

Color is not supported by terminal

exception progressive.exceptions.**LengthOverflowError**
Bases: *progressive.exceptions.ProgressiveException*

Terminal is not long enough to display hierarchy

exception progressive.exceptions.**ProgressiveException**
Bases: exceptions.Exception

Base class for exceptions raised by progressive

exception progressive.exceptions.**WidthOverflowError**
Bases: *progressive.exceptions.ProgressiveException*

Terminal is not wide enough for the bar attempting to be written

1.1.5 progressive.pretty module

1.1.6 progressive.tree module

class progressive.tree.**BarDescriptor**
Bases: dict

Bar descriptor

To be used in leaf of a tree describing a hierarchy for ProgressTree, e.g.,:

tree = {"Job": {"Task1": BarDescriptor(...)}, {"Task2":

{"Subtask1": BarDescriptor(...)},

},

}

Parameters

- **type** (*Bar|subclass of Bar*) – The type of Bar to use to display that leaf
- **value** (Value) – Amount to fill the progress bar vs. its max value
- **args** (*list*) – A list of args to instantiate type with
- **kwargs** (*dict*) – A dict of kwargs to instantiate type with

class progressive.tree.**ProgressTree**(*term=None*, *indent=4*)
Bases: object

Progress display for trees

For drawing a hierarchical progress view from a tree

Parameters

- **term** (*NoneType|blessings.Terminal*) – Terminal instance; if not given, will be created by the class
- **indent** (*int*) – The amount of indentation between each level in hierarchy

draw(*tree*, *bar_desc=None*, *save_cursor=True*, *flush=True*)
Draw tree to the terminal

Parameters

- **tree** (*dict*) – `tree` should be a tree representing a hierarchy; each key should be a string describing that hierarchy level and value should also be `dict` except for leaves which should be `BarDescriptors`. See `BarDescriptor` for a tree example.
- **bar_desc** (*BarDescriptor|NoneType*) – For describing non-leaf bars in that will be drawn from `tree`; certain attributes such as `value` and `kwargs["max_value"]` will of course be overridden if provided.
- **flush** (*bool*) – If this is set, output written will be flushed
- **save_cursor** (*bool*) – If this is set, cursor location will be saved before drawing; this will OVERWRITE a previous save, so be sure to set this accordingly (to your needs).

lines_required(*tree*, *count=0*)
Calculate number of lines required to draw `tree`

make_room(*tree*)
Clear lines in terminal below current cursor position as required

This is important to do before drawing to ensure sufficient room at the bottom of your terminal.

Parameters **tree** (*dict*) – tree as described in `BarDescriptor`

class progressive.tree.**Value**(*val=0*)
Bases: `object`

Container class for use with `BarDescriptor`

Should be used for `value` argument when initializing `BarDescriptor`, e.g., `BarDescriptor(type=..., value=Value(10))`

value

1.1.7 progressive.util module

progressive.util.**ensure**(*expr*, *exc*, **args*, ***kwargs*)

Raises **exc** – With `*args` and `**kwargs` if not `expr`

progressive.util.**floor**(*x*)
Returns the floor of `x` :returns: floor of `x` :rtype: int

progressive.util.**merge_dicts**(*dicts*, *deepcopy=False*)
Merges dicts

In case of key conflicts, the value kept will be from the latter dictionary in the list of dictionaries

Parameters

- **dicts** – [dict, . . .]
- **deepcopy** – deepcopy items within dicts

progressive.util.**u**(*s*)
Cast `s` as unicode string

This is a convenience function to make up for the fact that Python3 does not have a unicode() cast (for obvious reasons)

Return type unicode

Returns Equivalent of unicode(s) (at least I hope so)

Python Module Index

p

Index