```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import time

from max7219.font import DEFAULT_FONT
from max7219.rotate8x8 import rotate


class constants(object):
    MAX7219_REG_NOOP = 0x0
    MAX7219_REG_DIGIT0 = 0x1
    MAX7219_REG_DIGIT1 = 0x2
    MAX7219_REG_DIGIT2 = 0x3
    MAX7219_REG_DIGIT3 = 0x4
    MAX7219_REG_DIGIT4 = 0x5
    MAX7219_REG_DIGIT5 = 0x6
    MAX7219_REG_DIGIT6 = 0x7
    MAX7219_REG_DIGIT7 = 0x8
    MAX7219_REG_DECODEMODE = 0x9
    MAX7219_REG_INTENSITY = 0xA
    MAX7219_REG_SCANLIMIT = 0xB
    MAX7219_REG_SHUTDOWN = 0xC
    MAX7219_REG_DISPLAYTEST = 0xF


class device(object):
    """
    Base class for handling multiple cascaded MAX7219 devices.
    Callers should generally pick either the :py:class:`sevensegment` or
    :py:class:`matrix` subclasses instead depending on which application
    is required.

    A buffer is maintained which holds the bytes that will be cascaded
    every time :py:func:`flush` is called.
    """
    NUM_DIGITS = 8

    def __init__(self, cascaded=1, spi_bus=0, spi_device=0, vertical=False):
        """
        Constructor: `cascaded` should be the number of daisy-chained MAX7219
        devices that are connected. `vertical` should be set to True if
        the text should start from the header instead perpendicularly.
        """
        import spidev
        assert cascaded > 0, "Must have at least one device!"

        self._cascaded = cascaded
        self._buffer = [0] * self.NUM_DIGITS * self._cascaded
        self._spi = spidev.SpiDev()
        self._spi.open(spi_bus, spi_device)
        self._vertical = vertical

        self.command(constants.MAX7219_REG_SCANLIMIT, 7)    # show all 8 digits
        self.command(constants.MAX7219_REG_DECODEMODE, 0)   # use matrix (not digits)
        self.command(constants.MAX7219_REG_DISPLAYTEST, 0)  # no display test
        self.command(constants.MAX7219_REG_SHUTDOWN, 1)     # not shutdown mode
        self.brightness(7)                                  # intensity: range: 0..15
        self.clear()

    def command(self, register, data):
        """
        Sends a specific register some data, replicated for all cascaded
        devices
        """
        assert constants.MAX7219_REG_DECODEMODE <= register <= constants.MAX7219_REG_DISPLAYTEST
        self._write([register, data] * self._cascaded)

    def _write(self, data):
```

```
        """
        Send the bytes (which should comprise of alternating command,
        data values) over the SPI device.
        """
        self._spi.xfer2(list(data))

    def _values(self, position, buf):
        """
        A generator which yields the digit/column position and the data
        value from that position for each of the cascaded devices.
        """
        for deviceId in range(self._cascaded):
            yield position + constants.MAX7219_REG_DIGIT0
            yield buf[(deviceId * self.NUM_DIGITS) + position]

    def clear(self, deviceId=None):
        """
        Clears the buffer the given deviceId if specified (else clears all
        devices), and flushes.
        """
        assert not deviceId or 0 <= deviceId < self._cascaded, "Invalid deviceId:
{0}".format(deviceId)

        if deviceId is None:
            start = 0
            end = self._cascaded
        else:
            start = deviceId
            end = deviceId + 1

        for deviceId in range(start, end):
            for position in range(self.NUM_DIGITS):
                self.set_byte(deviceId,
                              position + constants.MAX7219_REG_DIGIT0,
                              0, redraw=False)

        self.flush()

    def _preprocess_buffer(self, buf):
        """
        Overload in subclass to provide custom behaviour: see
        matrix implementation for example.
        """
        return buf

    def flush(self):
        """
        For each digit/column, cascade out the contents of the buffer
        cells to the SPI device.
        """
        # Allow subclasses to pre-process the buffer: they shouldn't
        # alter it, so make a copy first.
        buf = self._preprocess_buffer(list(self._buffer))
        assert len(buf) == len(self._buffer), "Preprocessed buffer is wrong size"
        if self._vertical:
            tmp_buf = []
            for x in range(0, self._cascaded):
                tmp_buf += rotate(buf[x*8:x*8+8])
            buf = tmp_buf

        for posn in range(self.NUM_DIGITS):
            self._write(self._values(posn, buf))

    def brightness(self, intensity):
        """
        Sets the brightness level of all cascaded devices to the same
        intensity level, ranging from 0..15. Note that setting the brightness
        to a high level will draw more current, and may cause intermittent
        issues / crashes if the USB power source is insufficient.
```

```
        """
        assert 0 <= intensity < 16, "Invalid brightness: {0}".format(intensity)
        self.command(constants.MAX7219_REG_INTENSITY, intensity)

    def set_byte(self, deviceId, position, value, redraw=True):
        """
        Low level mechanism to set a byte value in the buffer array. If redraw
        is not suppled, or set to True, will force a redraw of _all_ buffer
        items: If you are calling this method rapidly/frequently (e.g in a
        loop), it would be more efficient to set to False, and when done,
        call :py:func:`flush`.

        Prefer to use the higher-level method calls in the subclasses below.
        """
        assert 0 <= deviceId < self._cascaded, "Invalid deviceId: {0}".format(deviceId)
        assert constants.MAX7219_REG_DIGIT0 <= position <= constants.MAX7219_REG_DIGIT7, "Invalid
digit/column: {0}".format(position)
        assert 0 <= value < 256, 'Value {0} outside range 0..255'.format(value)

        offset = (deviceId * self.NUM_DIGITS) + position - constants.MAX7219_REG_DIGIT0
        self._buffer[offset] = value

        if redraw:
            self.flush()

    def rotate_left(self, redraw=True):
        """
        Scrolls the buffer one column to the left. The data that scrolls off
        the left side re-appears at the right-most position. If redraw
        is not suppled, or left set to True, will force a redraw of _all_ buffer
        items
        """
        t = self._buffer[-1]
        for i in range((self.NUM_DIGITS * self._cascaded) - 1, 0, -1):
            self._buffer[i] = self._buffer[i - 1]
        self._buffer[0] = t
        if redraw:
            self.flush()

    def rotate_right(self, redraw=True):
        """
        Scrolls the buffer one column to the right. The data that scrolls off
        the right side re-appears at the left-most position. If redraw
        is not suppled, or left set to True, will force a redraw of _all_ buffer
        items
        """
        t = self._buffer[0]
        for i in range(0, (self.NUM_DIGITS * self._cascaded) - 1, 1):
            self._buffer[i] = self._buffer[i + 1]
        self._buffer[-1] = t
        if redraw:
            self.flush()

    def scroll_left(self, redraw=True):
        """
        Scrolls the buffer one column to the left. Any data that scrolls off
        the left side is lost and does not re-appear on the right. An empty
        column is inserted at the right-most position. If redraw
        is not suppled, or set to True, will force a redraw of _all_ buffer
        items
        """
        del self._buffer[0]
        self._buffer.append(0)
        if redraw:
            self.flush()

    def scroll_right(self, redraw=True):
        """
        Scrolls the buffer one column to the right. Any data that scrolls off
```

```
        the right side is lost and does not re-appear on the left. An empty
        column is inserted at the left-most position. If redraw
        is not suppled, or set to True, will force a redraw of _all_ buffer
        items
        """
        del self._buffer[-1]
        self._buffer.insert(0, 0)
        if redraw:
            self.flush()


class sevensegment(device):
    """
    Implementation of MAX7219 devices cascaded with a series of seven-segment
    LEDs. It provides a convenient method to write a number to a given device
    in octal, decimal or hex, flushed left/right with zero padding. Base 10
    numbers can be either integers or floating point (with the number of
    decimal points configurable).
    """
    _UNDEFINED = 0x08
    _RADIX = {8: 'o', 10: 'f', 16: 'x'}
    # Some letters cannot be represented by 7 segments, so dictionay lookup
    # will default to _UNDEFINED (an underscore) instead.
    _DIGITS = {
        ' ': 0x00,
        '-': 0x01,
        '_': 0x08,
        '0': 0x7e,
        '1': 0x30,
        '2': 0x6d,
        '3': 0x79,
        '4': 0x33,
        '5': 0x5b,
        '6': 0x5f,
        '7': 0x70,
        '8': 0x7f,
        '9': 0x7b,
        'a': 0x7d,
        'b': 0x1f,
        'c': 0x0d,
        'd': 0x3d,
        'e': 0x6f,
        'f': 0x47,
        'g': 0x7b,
        'h': 0x17,
        'i': 0x10,
        'j': 0x18,
        # 'k': cant represent
        'l': 0x06,
        # 'm': cant represent
        'n': 0x15,
        'o': 0x1d,
        'p': 0x67,
        'q': 0x73,
        'r': 0x05,
        's': 0x5b,
        't': 0x0f,
        'u': 0x1c,
        'v': 0x1c,
        # 'w': cant represent
        # 'x': cant represent
        'y': 0x3b,
        'z': 0x6d,
        'A': 0x77,
        'B': 0x7f,
        'C': 0x4e,
        'D': 0x7e,
        'E': 0x4f,
        'F': 0x47,
```

```
        'G': 0x5e,
        'H': 0x37,
        'I': 0x30,
        'J': 0x38,
        # 'K': cant represent
        'L': 0x0e,
        # 'M': cant represent
        'N': 0x76,
        'O': 0x7e,
        'P': 0x67,
        'Q': 0x73,
        'R': 0x46,
        'S': 0x5b,
        'T': 0x0f,
        'U': 0x3e,
        'V': 0x3e,
        # 'W': cant represent
        # 'X': cant represent
        'Y': 0x3b,
        'Z': 0x6d,
        ',': 0x80,
        '.': 0x80
    }

    def letter(self, deviceId, position, char, dot=False, redraw=True):
        """
        Looks up the most appropriate character representation for char
        from the digits table, and writes that bitmap value into the buffer
        at the given deviceId / position.
        """
        assert dot in [0, 1, False, True]
        value = self._DIGITS.get(str(char), self._UNDEFINED) | (dot << 7)
        self.set_byte(deviceId, position, value, redraw)

    def write_number(self, deviceId, value, base=10, decimalPlaces=0,
                     zeroPad=False, leftJustify=False):
        """
        Formats the value according to the parameters supplied, and displays
        on the specified device. If the formatted number is larger than
        8 digits, then an OverflowError is raised.
        """
        assert 0 <= deviceId < self._cascaded, "Invalid deviceId: {0}".format(deviceId)
        assert base in self._RADIX, "Invalid base: {0}".format(base)

        # Magic up a printf format string
        size = self.NUM_DIGITS
        formatStr = '%'

        if zeroPad:
            formatStr += '0'

        if decimalPlaces > 0:
            size += 1

        if leftJustify:
            size *= -1

        formatStr = '{fmt}{size}.{dp}{type}'.format(
                        fmt=formatStr, size=size, dp=decimalPlaces,
                        type=self._RADIX[base])

        position = constants.MAX7219_REG_DIGIT7
        strValue = formatStr % value

        # Go through each digit in the formatted string,
        # updating the buffer accordingly
        for char in strValue:

            if position < constants.MAX7219_REG_DIGIT0:
```

```
                self.clear(deviceId)
                raise OverflowError('{0} too large for display'.format(strValue))

            if char == '.':
                continue

            dp = (decimalPlaces > 0 and position == decimalPlaces + 1)
            self.letter(deviceId, position, char, dot=dp, redraw=False)
            position -= 1

        self.flush()

    def write_text(self, deviceId, text):
        """
        Outputs the text (as near as possible) on the specific device. If
        text is larger than 8 characters, then an OverflowError is raised.
        """
        assert 0 <= deviceId < self._cascaded, "Invalid deviceId: {0}".format(deviceId)
        if len(text) > 8:
            raise OverflowError('{0} too large for display'.format(text))
        for pos, char in enumerate(text.ljust(8)[::-1]):
            self.letter(deviceId, constants.MAX7219_REG_DIGIT0 + pos, char, redraw=False)

        self.flush()

    def show_message(self, text, delay=0.4):
        """
        Transitions the text message across the devices from left-to-right
        """
        # Add some spaces on (same number as cascaded devices) so that the
        # message scrolls off to the left completely.
        text += ' ' * self._cascaded * 8
        for value in text:
            time.sleep(delay)
            self.scroll_right(redraw=False)
            self._buffer[0] = self._DIGITS.get(value, self._UNDEFINED)
            self.flush()


class matrix(device):
    """
    Implementation of MAX7219 devices cascaded with a series of 8x8 LED
    matrix devices. It provides a convenient methods to write letters
    to specific devices, to scroll a large message from left-to-right, or
    to set specific pixels. It is assumed the matrices are linearly aligned.
    """

    _invert = 0
    _orientation = 0

    def letter(self, deviceId, asciiCode, font=None, redraw=True):
        """
        Writes the ASCII letter code to the given device in the specified font.
        """
        assert 0 <= asciiCode < 256

        if not font:
            font = DEFAULT_FONT

        col = constants.MAX7219_REG_DIGIT0
        for value in font[asciiCode]:
            if col > constants.MAX7219_REG_DIGIT7:
                self.clear(deviceId)
                raise OverflowError('Font for \'{0}\' too large for display'.format(asciiCode))

            self.set_byte(deviceId, col, value, redraw=False)
            col += 1

        if redraw:
```

```
            self.flush()

    def scroll_up(self, redraw=True):
        """
        Scrolls the underlying buffer (for all cascaded devices) up one pixel
        """
        self._buffer = [value >> 1 for value in self._buffer]
        if redraw:
            self.flush()

    def scroll_down(self, redraw=True):
        """
        Scrolls the underlying buffer (for all cascaded devices) down one pixel
        """
        self._buffer = [(value << 1) & 0xff for value in self._buffer]
        if redraw:
            self.flush()

    def show_message(self, text, font=None, delay=0.05, always_scroll=False):
        """
        Shows a message on the device. If it's longer then the total width
        (or always_scroll=True), it transitions the text message across the
        devices from right-to-left.
        """
        if not font:
            font = DEFAULT_FONT

        display_length = self.NUM_DIGITS * self._cascaded
        src = [c for ascii_code in text for c in font[ord(ascii_code)]]
        scroll = always_scroll or len(src) > display_length
        if scroll:
            # Add some spaces on (same number as cascaded devices) so that the
            # message scrolls off to the left completely.
            src += [c for ascii_code in ' ' * self._cascaded
                    for c in font[ord(ascii_code)]]
        else:
            # How much margin we need on the left so it's centered
            margin = int((display_length - len(src))/2)
            # Reset the buffer so no traces of the previous message are left
            self._buffer = [0] * display_length
        for pos, value in enumerate(src):
            if scroll:
                time.sleep(delay)
                self.scroll_left(redraw=False)
                self._buffer[-1] = value
                self.flush()
            else:
                self._buffer[margin+pos] = value
        if not scroll:
            self.flush()

    def pixel(self, x, y, value, redraw=True):
        """
        Sets (value = 1) or clears (value = 0) the pixel at the given
        co-ordinate. It may be more efficient to batch multiple pixel
        operations together with redraw=False, and then call
        :py:func:`flush` to redraw just once.
        """
        assert 0 <= x < len(self._buffer)
        assert 0 <= y < self.NUM_DIGITS

        if value:
            self._buffer[x] |= (1 << y)
        else:
            self._buffer[x] &= ~(1 << y)

        if redraw:
            self.flush()
```

```
    def _rotate(self, buf):
        """
        Rotates tiles in the buffer by the given orientation
        """
        result = []
        for i in range(0, self._cascaded * self.NUM_DIGITS, self.NUM_DIGITS):
            tile = buf[i:i + self.NUM_DIGITS]
            for _ in range(self._orientation // 90):
                tile = rotate(tile)

            result += tile

        return result

    def _preprocess_buffer(self, buf):
        """
        Inverts and/or orientates the buffer before flushing according to
        user set parameters
        """
        if self._invert:
            buf = [~x & 0xff for x in buf]

        if self._orientation:
            buf = self._rotate(buf)

        return super(matrix, self)._preprocess_buffer(buf)

    def invert(self, value, redraw=True):
        """
        Sets whether the display should be inverted or not when displaying
        letters.
        """
        assert value in [0, 1, False, True]

        self._invert = value
        if redraw:
            self.flush()

    def orientation(self, angle, redraw=True):
        """
        Sets the orientation (angle should be 0, 90, 180 or 270) at which
        the characters are displayed.
        """
        assert angle in [0, 90, 180, 270]

        self._orientation = angle
        if redraw:
            self.flush()
```