# -\*- coding: utf-8 -\*-

# Copyright (c) 2013, Michael Nooner

# All rights reserved.

# Redistribution and use in source and binary forms, with or without

# modification, are permitted provided that the following conditions are met:

# \* Redistributions of source code must retain the above copyright

# notice, this list of conditions and the following disclaimer.

# \* Redistributions in binary form must reproduce the above copyright

# notice, this list of conditions and the following disclaimer in the

# documentation and/or other materials provided with the distribution.

# \* Neither the name of the copyright holder nor the names of its

# contributors may be used to endorse or promote products derived from

# this software without specific prior written permission

# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"

# AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE

# IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE

# ARE DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY

# DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES

# (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;

# LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND

# ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT

# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS

# SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

"""This module is used to create QR Codes. It is designed to be as simple and

as possible. It does this by using sane defaults and autodetection to make

creating a QR Code very simple.

It is recommended that you use the :func:`pyqrcode.create` function to build the

QRCode object. This results in cleaner looking code.

Examples:

>>> import pyqrcode

>>> import sys

>>> url = pyqrcode.create('http://uca.edu')

>>> url.svg(sys.stdout, scale=1)

>>> url.svg('uca.svg', scale=4)

>>> number = pyqrcode.create(123456789012345)

>>> number.png('big-number.png')

"""

#Imports required for 2.7 support

from \_\_future\_\_ import absolute\_import, division, print\_function, with\_statement, unicode\_literals

from . import tables

from . import builder as builder

try:

str = unicode # Python 2

except NameError:

pass

def create(content, error='H', version=None, mode=None, encoding='utf-8'):

"""When creating a QR code only the content to be encoded is required,

all the other properties of the code will be guessed based on the

contents given. This function will return a :class:`QRCode` object.

Unless you are familiar with QR code's inner workings

it is recommended that you just specify the \*content\* and nothing else.

However, there are cases where you may want to specify the various

properties of the created code manually, this is what the other

parameters do. Below, you will find a lengthy explanation of what

each parameter is for. Note, the parameter names and values are taken

directly from the standards. You may need to familiarize yourself

with the terminology of QR codes for the names and their values to

make sense.

The \*error\* parameter sets the error correction level of the code. There

are four levels defined by the standard. The first is level 'L' which

allows for 7% of the code to be corrected. Second, is level 'M' which

allows for 15% of the code to be corrected. Next, is level 'Q' which

is the most common choice for error correction, it allow 25% of the

code to be corrected. Finally, there is the highest level 'H' which

allows for 30% of the code to be corrected. There are several ways to

specify this parameter, you can use an upper or lower case letter,

a float corresponding to the percentage of correction, or a string

containing the percentage. See tables.modes for all the possible

values. By default this parameter is set to 'H' which is the highest

possible error correction, but it has the smallest available data

capacity.

The \*version\* parameter specifies the size and data capacity of the

code. Versions are any integer between 1 and 40. Where version 1 is

the smallest QR code, and version 40 is the largest. If this parameter

is left unspecified, then the contents and error correction level will

be used to guess the smallest possible QR code version that the

content will fit inside of. You may want to specify this parameter

for consistency when generating several QR codes with varying amounts

of data. That way all of the generated codes would have the same size.

The \*mode\* parameter specifies how the contents will be encoded. By

default, the best possible encoding for the contents is guessed. There

are four possible encoding methods. First, is 'numeric' which is

used to encode integer numbers. Next, is 'alphanumeric' which is

used to encode some ASCII characters. This mode uses only a limited

set of characters. Most problematic is that it can only use upper case

English characters, consequently, the content parameter will be

subjected to str.upper() before encoding. See tables.ascii\_codes for

a complete list of available characters. We then have 'binary' encoding

which just encodes the bytes directly into the QR code (this encoding

is the least efficient). Finally, there is 'kanji' encoding (i.e.

Japanese characters), this encoding is unimplemented at this time.

The \*encoding\* parameter specifies how the content will be interpreted.

This parameter only matters if the \*content\* is a string, unicode, or

byte array type. This parameter must be a valid encoding string. It will

be passed the \*content\*'s encode/decode methods.

"""

return QRCode(content, error, version, mode, encoding)

class QRCode:

"""This class represents a QR code. To use this class simply give the

constructor a string representing the data to be encoded, it will then

build a code in memory. You can then save it in various formats. Note,

codes can be written out as PNG files but this requires the PyPNG module.

You can find the PyPNG module at http://packages.python.org/pypng/.

Examples:

>>> from pyqrcode import QRCode

>>> import sys

>>> url = QRCode('http://uca.edu')

>>> url.svg(sys.stdout, scale=1)

>>> url.svg('uca.svg', scale=4)

>>> number = QRCode(123456789012345)

>>> number.png('big-number.png')

.. note::

For what all of the parameters do, see the :func:`pyqrcode.create`

function.

"""

def \_\_init\_\_(self, content, error='H', version=None, mode=None,

encoding='utf-8'):

#Store the encoding for use later

if encoding is None:

encoding = 'utf-8'

self.encoding = encoding

if version is not None:

if 1 <= version <= 40:

self.version = version

else:

raise ValueError("Illegal version {0}, version must be between "

"1 and 40.".format(version))

#Decode a 'byte array' contents into a string format

if isinstance(content, bytes):

self.data = content.decode(encoding)

#Encode a string an encoding

elif hasattr(content, 'encode'):

#Try encoding using the given value

if encoding is not None:

self.data = content.encode(encoding)

else:

# Try to use standard-conforming encoding

try:

self.data = content.encode('iso-8859-1')

self.encoding = 'iso-8859-1'

except UnicodeError:

self.data = content.encode('utf-8')

self.encoding = 'utf-8'

#The contents are not a byte array or string, so

#try naively converting to a string representation.

else:

#Python2 vs. Python3 compatibility

try:

self.data = unicode(content)

except NameError:

self.data = str(content)

#Guess the mode of the code, this will also be used for

#error checking

guessed\_content\_type = self.\_detect\_content\_type(self.data)

#Force a passed in mode to be lowercase

if hasattr(mode, 'lower'):

mode = mode.lower()

#Check that the mode parameter is compatible with the contents

if mode is None:

#Use the guessed mode

self.mode = guessed\_content\_type

self.mode\_num = tables.modes[self.mode]

elif mode not in tables.modes.keys():

#Unknown mode

raise ValueError('{0} is not a valid mode.'.format(mode))

elif guessed\_content\_type == 'binary' and \

tables.modes[mode] != tables.modes['binary']:

#Binary is only guessed as a last resort, if the

#passed in mode is not binary the data won't encode

raise ValueError('The content provided cannot be encoded with '

'the mode {}, it can only be encoded as '

'binary.'.format(mode))

elif tables.modes[mode] == tables.modes['numeric'] and \

guessed\_content\_type != 'numeric':

#If numeric encoding is requested make sure the data can

#be encoded in that format

raise ValueError('The content cannot be encoded as numeric.')

else:

#The data should encode with the passed in mode

self.mode = mode

self.mode\_num = tables.modes[self.mode]

#Check that the user passed in a valid error level

if error in tables.error\_level.keys():

self.error = tables.error\_level[error]

else:

raise ValueError('{0} is not a valid error '

'level.'.format(error))

#Guess the "best" version

self.version = self.\_pick\_best\_fit(self.data)

#If the user supplied a version, then check that it has

#sufficient data capacity for the contents passed in

if version:

if version >= self.version:

self.version = version

else:

raise ValueError('The data will not fit inside a version {} '

'code with the given encoding and error '

'level (the code must be at least a '

'version {}).'.format(version, self.version))

#Build the QR code

self.builder = builder.QRCodeBuilder(data=self.data,

version=self.version,

mode=self.mode,

error=self.error)

#Save the code for easier reference

self.code = self.builder.code

def \_\_str\_\_(self):

return repr(self)

def \_\_unicode\_\_(self):

return self.\_\_repr\_\_()

def \_\_repr\_\_(self):

return "QRCode(content={0}, error='{1}', version={2}, mode='{3}')" \

.format(repr(self.data), self.error, self.version, self.mode)

def \_detect\_content\_type(self, content):

"""This method tries to auto-detect the type of the data. It first

tries to see if the data is a valid integer, in which case it returns

numeric. Next, it tests the data to see if it is 'alphanumeric.' QR

Codes use a special table with very limited range of ASCII characters.

The code's data is tested to make sure it fits inside this limited

range. If all else fails, the data is determined to be of type

'binary.'

Note, encoding 'kanji' and ECI is not yet implemented.

"""

#See if the data is an integer

try:

test = int(content)

return 'numeric'

except:

#Data is not numeric, this is not an error

pass

#See if that data is alphanumeric based on the standards

#special ASCII table

valid\_characters = ''.join(tables.ascii\_codes.keys())

#Force the characters into a byte array

valid\_characters = valid\_characters.encode('ASCII')

try:

if all(map(lambda x: x in valid\_characters, content)):

return 'alphanumeric'

except TypeError:

#This occurs if the content does not contain ASCII characters.

#Since the whole point of the if statement is to look for ACII

#characters, the resulting mode should be binary.

#Hence, this is not an error.

pass

#All of the tests failed. The content can only be binary.

return 'binary'

def \_pick\_best\_fit(self, content):

"""This method return the smallest possible QR code version number

that will fit the specified data with the given error level.

"""

for version in range(1,41):

#Get the maximum possible capacity

capacity = tables.data\_capacity[version][self.error][self.mode\_num]

#Check the capacity

if capacity >= len(content):

return version

raise ValueError('The data will not fit in any QR code version '

'with the given encoding and error level.')

def get\_png\_size(self, scale=1, quiet\_zone=4):

"""This is method helps users determine what \*scale\* to use when

creating a PNG of this QR code. It is meant mostly to be used in the

console to help the user determine the pixel size of the code

using various scales.

This method will return an integer representing the width and height of

the QR code in pixels, as if it was drawn using the given \*scale\*.

Because QR codes are square, the number represents both the width

and height dimensions.

The \*quiet\_zone\* parameter sets how wide the quiet zone around the code

should be. According to the standard this should be 4 modules. It is

left settable because such a wide quiet zone is unnecessary in many

applications where the QR code is not being printed.

Example:

>>> code = pyqrcode.QRCode("I don't like spam!")

>>> print(code.get\_png\_size(1))

31

>>> print(code.get\_png\_size(5))

155

"""

return builder.\_get\_png\_size(self.version, scale, quiet\_zone)

def show(self, wait=1.2, scale=10, module\_color=(0, 0, 0, 255),

background=(255, 255, 255, 255), quiet\_zone=4):

"""Displays this QR code.

This method is mainly intended for debugging purposes.

This method saves the output of the `png` method (with a default

scaling factor of 10) to a temporary file and opens it with the

standard PNG viewer application or within the standard webbrowser. The

temporary file is deleted afterwards.

If this method does not show any result, try to increase the `wait`

parameter. This parameter specifies the time in seconds to wait till

the temporary file is deleted. Note, that this method does not return

until the provided amount of seconds (default: 1.2) has passed.

The other parameters are simply passed on to the `png` method.

"""

import os

import time

import tempfile

import webbrowser

try: # Python 2

from urlparse import urljoin

from urllib import pathname2url

except ImportError: # Python 3

from urllib.parse import urljoin

from urllib.request import pathname2url

f = tempfile.NamedTemporaryFile('wb', suffix='.png', delete=False)

self.png(f, scale=scale, module\_color=module\_color,

background=background, quiet\_zone=quiet\_zone)

f.close()

webbrowser.open\_new\_tab(urljoin('file:', pathname2url(f.name)))

time.sleep(wait)

os.unlink(f.name)

def png(self, file, scale=1, module\_color=(0, 0, 0, 255),

background=(255, 255, 255, 255), quiet\_zone=4):

"""This method writes the QR code out as an PNG image. The resulting

PNG has a bit depth of 1. The file parameter is used to specify where

to write the image to. It can either be an writable stream or a

file path.

.. note::

This method depends on the pypng module to actually create the

PNG file.

This method will write the given \*file\* out as a PNG file. The file

can be either a string file path, or a writable stream. The file

will not be automatically closed if a stream is given.

The \*scale\* parameter sets how large to draw a single module. By

default one pixel is used to draw a single module. This may make the

code too small to be read efficiently. Increasing the scale will make

the code larger. Only integer scales are usable. This method will

attempt to coerce the parameter into an integer (e.g. 2.5 will become 2,

and '3' will become 3).

The \*module\_color\* parameter sets what color to use for the encoded

modules (the black part on most QR codes). The \*background\* parameter

sets what color to use for the background (the white part on most

QR codes). If either parameter is set, then both must be

set or a ValueError is raised. Colors should be specified as either

a list or a tuple of length 3 or 4. The components of the list must

be integers between 0 and 255. The first three member give the RGB

color. The fourth member gives the alpha component, where 0 is

transparent and 255 is opaque. Note, many color

combinations are unreadable by scanners, so be judicious.

The \*quiet\_zone\* parameter sets how wide the quiet zone around the code

should be. According to the standard this should be 4 modules. It is

left settable because such a wide quiet zone is unnecessary in many

applications where the QR code is not being printed.

Example:

>>> code = pyqrcode.create('Are you suggesting coconuts migrate?')

>>> code.png('swallow.png', scale=5)

>>> code.png('swallow.png', scale=5,

module\_color=(0x66, 0x33, 0x0), #Dark brown

background=(0xff, 0xff, 0xff, 0x88)) #50% transparent white

"""

builder.\_png(self.code, self.version, file, scale,

module\_color, background, quiet\_zone)

def svg(self, file, scale=1, module\_color='#000', background=None,

quiet\_zone=4, xmldecl=True, svgns=True, title=None,

svgclass='pyqrcode', lineclass='pyqrline', omithw=False,

debug=False):

"""This method writes the QR code out as an SVG document. The

code is drawn by drawing only the modules corresponding to a 1. They

are drawn using a line, such that contiguous modules in a row

are drawn with a single line.

The \*file\* parameter is used to specify where to write the document

to. It can either be a writable stream or a file path.

The \*scale\* parameter sets how large to draw

a single module. By default one pixel is used to draw a single

module. This may make the code too small to be read efficiently.

Increasing the scale will make the code larger. Unlike the png() method,

this method will accept fractional scales (e.g. 2.5).

Note, three things are done to make the code more appropriate for

embedding in a HTML document. The "white" part of the code is actually

transparent. The code itself has a class given by \*svgclass\* parameter.

The path making up the QR code uses the class set using the \*lineclass\*.

These should make the code easier to style using CSS.

By default the output of this function is a complete SVG document. If

only the code itself is desired, set the \*xmldecl\* to false. This will

result in a fragment that contains only the "drawn" portion of the code.

Likewise, you can set the \*title\* of the document. The SVG name space

attribute can be suppressed by setting \*svgns\* to False.

When True the \*omithw\* indicates if width and height attributes should

be omitted. If these attributes are omitted, a ``viewBox`` attribute

will be added to the document.

You can also set the colors directly using the \*module\_color\* and

\*background\* parameters. The \*module\_color\* parameter sets what color to

use for the data modules (the black part on most QR codes). The

\*background\* parameter sets what color to use for the background (the

white part on most QR codes). The parameters can be set to any valid

SVG or HTML color. If the background is set to None, then no background

will be drawn, i.e. the background will be transparent. Note, many color

combinations are unreadable by scanners, so be careful.

The \*quiet\_zone\* parameter sets how wide the quiet zone around the code

should be. According to the standard this should be 4 modules. It is

left settable because such a wide quiet zone is unnecessary in many

applications where the QR code is not being printed.

Example:

>>> code = pyqrcode.create('Hello. Uhh, can we have your liver?')

>>> code.svg('live-organ-transplants.svg', 3.6)

>>> code.svg('live-organ-transplants.svg', scale=4,

module\_color='brown', background='0xFFFFFF')

"""

builder.\_svg(self.code, self.version, file, scale=scale,

module\_color=module\_color, background=background,

quiet\_zone=quiet\_zone, xmldecl=xmldecl, svgns=svgns,

title=title, svgclass=svgclass, lineclass=lineclass,

omithw=omithw, debug=debug)

def eps(self, file, scale=1, module\_color=(0, 0, 0),

background=None, quiet\_zone=4):

"""This method writes the QR code out as an EPS document. The

code is drawn by only writing the data modules corresponding to a 1.

They are drawn using a line, such that contiguous modules in a row

are drawn with a single line.

The \*file\* parameter is used to specify where to write the document

to. It can either be a writable (text) stream or a file path.

The \*scale\* parameter sets how large to draw a single module. By

default one point (1/72 inch) is used to draw a single module. This may

make the code to small to be read efficiently. Increasing the scale

will make the code larger. This method will accept fractional scales

(e.g. 2.5).

The \*module\_color\* parameter sets the color of the data modules. The

\*background\* parameter sets the background (page) color to use. They

are specified as either a triple of floats, e.g. (0.5, 0.5, 0.5), or a

triple of integers, e.g. (128, 128, 128). The default \*module\_color\* is

black. The default \*background\* color is no background at all.

The \*quiet\_zone\* parameter sets how large to draw the border around

the code. As per the standard, the default value is 4 modules.

Examples:

>>> qr = pyqrcode.create('Hello world')

>>> qr.eps('hello-world.eps', scale=2.5, module\_color='#36C')

>>> qr.eps('hello-world2.eps', background='#eee')

>>> out = io.StringIO()

>>> qr.eps(out, module\_color=(.4, .4, .4))

"""

builder.\_eps(self.code, self.version, file, scale, module\_color,

background, quiet\_zone)

def terminal(self, module\_color='default', background='reverse',

quiet\_zone=4):

"""This method returns a string containing ASCII escape codes,

such that if printed to a compatible terminal, it will display

a vaild QR code. The code is printed using ASCII escape

codes that alter the coloring of the background.

The \*module\_color\* parameter sets what color to

use for the data modules (the black part on most QR codes).

Likewise, the \*background\* parameter sets what color to use

for the background (the white part on most QR codes).

There are two options for colors. The first, and most widely

supported, is to use the 8 or 16 color scheme. This scheme uses

eight to sixteen named colors. The following colors are

supported the most widely supported: black, red, green,

yellow, blue, magenta, and cyan. There are an some additional

named colors that are supported by most terminals: light gray,

dark gray, light red, light green, light blue, light yellow,

light magenta, light cyan, and white.

There are two special named colors. The first is the

"default" color. This color is the color the background of

the terminal is set to. The next color is the "reverse"

color. This is not really a color at all but a special

property that will reverse the current color. These two colors

are the default values for \*module\_color\* and \*background\*

respectively. These values should work on most terminals.

Finally, there is one more way to specify the color. Some

terminals support 256 colors. The actual colors displayed in the

terminal is system dependent. This is the least transportable option.

To use the 256 color scheme set \*module\_color\* and/or

\*background\* to a number between 0 and 256.

The \*quiet\_zone\* parameter sets how wide the quiet zone around the code

should be. According to the standard this should be 4 modules. It is

left settable because such a wide quiet zone is unnecessary in many

applications.

Example:

>>> code = pyqrcode.create('Example')

>>> text = code.terminal()

>>> print(text)

"""

return builder.\_terminal(self.code, module\_color, background,

quiet\_zone)

def text(self, quiet\_zone=4):

"""This method returns a string based representation of the QR code.

The data modules are represented by 1's and the background modules are

represented by 0's. The main purpose of this method is to allow a user

to write their own renderer.

The \*quiet\_zone\* parameter sets how wide the quiet zone around the code

should be. According to the standard this should be 4 modules. It is

left settable because such a wide quiet zone is unnecessary in many

applications.

Example:

>>> code = pyqrcode.create('Example')

>>> text = code.text()

>>> print(text)

"""

return builder.\_text(self.code, quiet\_zone)