# -\*- 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 does the actual generation of the QR codes. The QRCodeBuilder

builds the code. While the various output methods draw the code into a file.

"""

#Imports required for 2.7 support

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

from . import tables as tables

import io

import itertools

class QRCodeBuilder:

"""This class generates a QR code based on the standard. It is meant to

be used internally, not by users!!!

This class implements the tutorials found at:

\* http://www.thonky.com/qr-code-tutorial/

\* http://www.matchadesign.com/blog/qr-code-demystified-part-6/

This class also uses the standard, which can be read online at:

http://raidenii.net/files/datasheets/misc/qr\_code.pdf

Test codes were tested against:

http://zxing.org/w/decode.jspx

Also, reference codes were generated at:

http://www.morovia.com/free-online-barcode-generator/qrcode-maker.php

QR code Debugger:

http://qrlogo.kaarposoft.dk/qrdecode.html

"""

def \_\_init\_\_(self, data, version, mode, error):

"""See :py:class:`pyqrcode.QRCode` for information on the parameters."""

#Set what data we are going to use to generate

#the QR code

self.data = data

#Check that the user passed in a valid mode

if mode in tables.modes.keys():

self.mode = tables.modes[mode]

else:

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

if 1 <= version <= 40:

self.version = version

else:

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

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

#Look up the proper row for error correction code words

self.error\_code\_words = tables.eccwbi[version][self.error]

#This property will hold the binary string as it is built

self.buffer = io.StringIO()

#Create the binary data block

self.add\_data()

#Create the actual QR code

self.make\_code()

def grouper(self, n, iterable, fillvalue=None):

"""This generator yields a set of tuples, where the

iterable is broken into n sized chunks. If the

iterable is not evenly sized then fillvalue will

be appended to the last tuple to make up the difference.

This function is copied from the standard docs on

itertools.

"""

args = [iter(iterable)] \* n

if hasattr(itertools, 'zip\_longest'):

return itertools.zip\_longest(\*args, fillvalue=fillvalue)

return itertools.izip\_longest(\*args, fillvalue=fillvalue)

def binary\_string(self, data, length):

"""This method returns a string of length n that is the binary

representation of the given data. This function is used to

basically create bit fields of a given size.

"""

return '{{0:0{0}b}}'.format(length).format(int(data))

def get\_data\_length(self):

"""QR codes contain a "data length" field. This method creates this

field. A binary string representing the appropriate length is

returned.

"""

#The "data length" field varies by the type of code and its mode.

#discover how long the "data length" field should be.

if 1 <= self.version <= 9:

max\_version = 9

elif 10 <= self.version <= 26:

max\_version = 26

elif 27 <= self.version <= 40:

max\_version = 40

data\_length = tables.data\_length\_field[max\_version][self.mode]

length\_string = self.binary\_string(len(self.data), data\_length)

if len(length\_string) > data\_length:

raise ValueError('The supplied data will not fit '

'within this version of a QRCode.')

return length\_string

def encode(self):

"""This method encodes the data into a binary string using

the appropriate algorithm specified by the mode.

"""

if self.mode == tables.modes['alphanumeric']:

encoded = self.encode\_alphanumeric()

elif self.mode == tables.modes['numeric']:

encoded = self.encode\_numeric()

elif self.mode == tables.modes['binary']:

encoded = self.encode\_bytes()

else:

raise ValueError('This mode is not yet implemented.')

return encoded

def encode\_alphanumeric(self):

"""This method encodes the QR code's data if its mode is

alphanumeric. It returns the data encoded as a binary string.

"""

#Convert the string to upper case

self.data = self.data.upper()

#Change the data such that it uses a QR code ascii table

ascii = []

for char in self.data:

if isinstance(char, int):

ascii.append(tables.ascii\_codes[chr(char)])

else:

ascii.append(tables.ascii\_codes[char])

#Now perform the algorithm that will make the ascii into bit fields

with io.StringIO() as buf:

for (a,b) in self.grouper(2, ascii):

if b is not None:

buf.write(self.binary\_string((45\*a)+b, 11))

else:

#This occurs when there is an odd number

#of characters in the data

buf.write(self.binary\_string(a, 6))

#Return the binary string

return buf.getvalue()

def encode\_numeric(self):

"""This method encodes the QR code's data if its mode is

numeric. It returns the data encoded as a binary string.

"""

with io.StringIO() as buf:

#Break the number into groups of three digits

for triplet in self.grouper(3, self.data):

number = ''

for digit in triplet:

#Only build the string if digit is not None

if digit:

number = ''.join([number, str(digit)])

else:

break

#If the number is one digits, make a 4 bit field

if len(number) == 1:

bin = self.binary\_string(number, 4)

#If the number is two digits, make a 7 bit field

elif len(number) == 2:

bin = self.binary\_string(number, 7)

#Three digit numbers use a 10 bit field

else:

bin = self.binary\_string(number, 10)

buf.write(bin)

return buf.getvalue()

def encode\_bytes(self):

"""This method encodes the QR code's data if its mode is

8 bit mode. It returns the data encoded as a binary string.

"""

with io.StringIO() as buf:

for char in self.data:

if not isinstance(char, int):

buf.write('{{0:0{0}b}}'.format(8).format(ord(char)))

else:

buf.write('{{0:0{0}b}}'.format(8).format(char))

return buf.getvalue()

def add\_data(self):

"""This function properly constructs a QR code's data string. It takes

into account the interleaving pattern required by the standard.

"""

#Encode the data into a QR code

self.buffer.write(self.binary\_string(self.mode, 4))

self.buffer.write(self.get\_data\_length())

self.buffer.write(self.encode())

#Fix for issue #3: https://github.com/mnooner256/pyqrcode/issues/3#

#I was performing the terminate\_bits() part in the encoding.

#As per the standard, terminating bits are only supposed to

#be added after the bit stream is complete. I took that to

#mean after the encoding, but actually it is after the entire

#bit stream has been constructed.

bits = self.terminate\_bits(self.buffer.getvalue())

if bits is not None:

self.buffer.write(bits)

#delimit\_words and add\_words can return None

add\_bits = self.delimit\_words()

if add\_bits:

self.buffer.write(add\_bits)

fill\_bytes = self.add\_words()

if fill\_bytes:

self.buffer.write(fill\_bytes)

#Get a numeric representation of the data

data = [int(''.join(x),2)

for x in self.grouper(8, self.buffer.getvalue())]

#This is the error information for the code

error\_info = tables.eccwbi[self.version][self.error]

#This will hold our data blocks

data\_blocks = []

#This will hold our error blocks

error\_blocks = []

#Some codes have the data sliced into two different sized blocks

#for example, first two 14 word sized blocks, then four 15 word

#sized blocks. This means that slicing size can change over time.

data\_block\_sizes = [error\_info[2]] \* error\_info[1]

if error\_info[3] != 0:

data\_block\_sizes.extend([error\_info[4]] \* error\_info[3])

#For every block of data, slice the data into the appropriate

#sized block

current\_byte = 0

for n\_data\_blocks in data\_block\_sizes:

data\_blocks.append(data[current\_byte:current\_byte+n\_data\_blocks])

current\_byte += n\_data\_blocks

#I am not sure about the test after the "and". This was added to

#fix a bug where after delimit\_words padded the bit stream, a zero

#byte ends up being added. After checking around, it seems this extra

#byte is supposed to be chopped off, but I cannot find that in the

#standard! I am adding it to solve the bug, I believe it is correct.

if current\_byte < len(data):

raise ValueError('Too much data for this code version.')

#DEBUG CODE!!!!

#Print out the data blocks

#print('Data Blocks:\n{0}'.format(data\_blocks))

#Calculate the error blocks

for n, block in enumerate(data\_blocks):

error\_blocks.append(self.make\_error\_block(block, n))

#DEBUG CODE!!!!

#Print out the error blocks

#print('Error Blocks:\n{0}'.format(error\_blocks))

#Buffer we will write our data blocks into

data\_buffer = io.StringIO()

#Add the data blocks

#Write the buffer such that: block 1 byte 1, block 2 byte 1, etc.

largest\_block = max(error\_info[2], error\_info[4])+error\_info[0]

for i in range(largest\_block):

for block in data\_blocks:

if i < len(block):

data\_buffer.write(self.binary\_string(block[i], 8))

#Add the error code blocks.

#Write the buffer such that: block 1 byte 1, block 2 byte 2, etc.

for i in range(error\_info[0]):

for block in error\_blocks:

data\_buffer.write(self.binary\_string(block[i], 8))

self.buffer = data\_buffer

def terminate\_bits(self, payload):

"""This method adds zeros to the end of the encoded data so that the

encoded data is of the correct length. It returns a binary string

containing the bits to be added.

"""

data\_capacity = tables.data\_capacity[self.version][self.error][0]

if len(payload) > data\_capacity:

raise ValueError('The supplied data will not fit '

'within this version of a QR code.')

#We must add up to 4 zeros to make up for any shortfall in the

#length of the data field.

if len(payload) == data\_capacity:

return None

elif len(payload) <= data\_capacity-4:

bits = self.binary\_string(0,4)

else:

#Make up any shortfall need with less than 4 zeros

bits = self.binary\_string(0, data\_capacity - len(payload))

return bits

def delimit\_words(self):

"""This method takes the existing encoded binary string

and returns a binary string that will pad it such that

the encoded string contains only full bytes.

"""

bits\_short = 8 - (len(self.buffer.getvalue()) % 8)

#The string already falls on an byte boundary do nothing

if bits\_short == 0 or bits\_short == 8:

return None

else:

return self.binary\_string(0, bits\_short)

def add\_words(self):

"""The data block must fill the entire data capacity of the QR code.

If we fall short, then we must add bytes to the end of the encoded

data field. The value of these bytes are specified in the standard.

"""

data\_blocks = len(self.buffer.getvalue()) // 8

total\_blocks = tables.data\_capacity[self.version][self.error][0] // 8

needed\_blocks = total\_blocks - data\_blocks

if needed\_blocks == 0:

return None

#This will return item1, item2, item1, item2, etc.

block = itertools.cycle(['11101100', '00010001'])

#Create a string of the needed blocks

return ''.join([next(block) for x in range(needed\_blocks)])

def \_fix\_exp(self, exponent):

"""Makes sure the exponent ranges from 0 to 255."""

#return (exponent % 256) + (exponent // 256)

return exponent % 255

def make\_error\_block(self, block, block\_number):

"""This function constructs the error correction block of the

given data block. This is \*very complicated\* process. To

understand the code you need to read:

\* http://www.thonky.com/qr-code-tutorial/part-2-error-correction/

\* http://www.matchadesign.com/blog/qr-code-demystified-part-4/

"""

#Get the error information from the standards table

error\_info = tables.eccwbi[self.version][self.error]

#This is the number of 8-bit words per block

if block\_number < error\_info[1]:

code\_words\_per\_block = error\_info[2]

else:

code\_words\_per\_block = error\_info[4]

#This is the size of the error block

error\_block\_size = error\_info[0]

#Copy the block as the message polynomial coefficients

mp\_co = block[:]

#Add the error blocks to the message polynomial

mp\_co.extend([0] \* (error\_block\_size))

#Get the generator polynomial

generator = tables.generator\_polynomials[error\_block\_size]

#This will hold the temporary sum of the message coefficient and the

#generator polynomial

gen\_result = [0] \* len(generator)

#Go through every code word in the block

for i in range(code\_words\_per\_block):

#Get the first coefficient from the message polynomial

coefficient = mp\_co.pop(0)

#Skip coefficients that are zero

if coefficient == 0:

continue

else:

#Turn the coefficient into an alpha exponent

alpha\_exp = tables.galois\_antilog[coefficient]

#Add the alpha to the generator polynomial

for n in range(len(generator)):

gen\_result[n] = alpha\_exp + generator[n]

if gen\_result[n] > 255:

gen\_result[n] = gen\_result[n] % 255

#Convert the alpha notation back into coefficients

gen\_result[n] = tables.galois\_log[gen\_result[n]]

#XOR the sum with the message coefficients

mp\_co[n] = gen\_result[n] ^ mp\_co[n]

#Pad the end of the error blocks with zeros if needed

if len(mp\_co) < code\_words\_per\_block:

mp\_co.extend([0] \* (code\_words\_per\_block - len(mp\_co)))

return mp\_co

def make\_code(self):

"""This method returns the best possible QR code."""

from copy import deepcopy

#Get the size of the underlying matrix

matrix\_size = tables.version\_size[self.version]

#Create a template matrix we will build the codes with

row = [' ' for x in range(matrix\_size)]

template = [deepcopy(row) for x in range(matrix\_size)]

#Add mandatory information to the template

self.add\_detection\_pattern(template)

self.add\_position\_pattern(template)

self.add\_version\_pattern(template)

#Create the various types of masks of the template

self.masks = self.make\_masks(template)

self.best\_mask = self.choose\_best\_mask()

self.code = self.masks[self.best\_mask]

def add\_detection\_pattern(self, m):

"""This method add the detection patterns to the QR code. This lets

the scanner orient the pattern. It is required for all QR codes.

The detection pattern consists of three boxes located at the upper

left, upper right, and lower left corners of the matrix. Also, two

special lines called the timing pattern is also necessary. Finally,

a single black pixel is added just above the lower left black box.

"""

#Draw outer black box

for i in range(7):

inv = -(i+1)

for j in [0,6,-1,-7]:

m[j][i] = 1

m[i][j] = 1

m[inv][j] = 1

m[j][inv] = 1

#Draw inner white box

for i in range(1, 6):

inv = -(i+1)

for j in [1, 5, -2, -6]:

m[j][i] = 0

m[i][j] = 0

m[inv][j] = 0

m[j][inv] = 0

#Draw inner black box

for i in range(2, 5):

for j in range(2, 5):

inv = -(i+1)

m[i][j] = 1

m[inv][j] = 1

m[j][inv] = 1

#Draw white border

for i in range(8):

inv = -(i+1)

for j in [7, -8]:

m[i][j] = 0

m[j][i] = 0

m[inv][j] = 0

m[j][inv] = 0

#To keep the code short, it draws an extra box

#in the lower right corner, this removes it.

for i in range(-8, 0):

for j in range(-8, 0):

m[i][j] = ' '

#Add the timing pattern

bit = itertools.cycle([1,0])

for i in range(8, (len(m)-8)):

b = next(bit)

m[i][6] = b

m[6][i] = b

#Add the extra black pixel

m[-8][8] = 1

def add\_position\_pattern(self, m):

"""This method draws the position adjustment patterns onto the QR

Code. All QR code versions larger than one require these special boxes

called position adjustment patterns.

"""

#Version 1 does not have a position adjustment pattern

if self.version == 1:

return

#Get the coordinates for where to place the boxes

coordinates = tables.position\_adjustment[self.version]

#Get the max and min coordinates to handle special cases

min\_coord = coordinates[0]

max\_coord = coordinates[-1]

#Draw a box at each intersection of the coordinates

for i in coordinates:

for j in coordinates:

#Do not draw these boxes because they would

#interfere with the detection pattern

if (i == min\_coord and j == min\_coord) or \

(i == min\_coord and j == max\_coord) or \

(i == max\_coord and j == min\_coord):

continue

#Center black pixel

m[i][j] = 1

#Surround the pixel with a white box

for x in [-1,1]:

m[i+x][j+x] = 0

m[i+x][j] = 0

m[i][j+x] = 0

m[i-x][j+x] = 0

m[i+x][j-x] = 0

#Surround the white box with a black box

for x in [-2,2]:

for y in [0,-1,1]:

m[i+x][j+x] = 1

m[i+x][j+y] = 1

m[i+y][j+x] = 1

m[i-x][j+x] = 1

m[i+x][j-x] = 1

def add\_version\_pattern(self, m):

"""For QR codes with a version 7 or higher, a special pattern

specifying the code's version is required.

For further information see:

http://www.thonky.com/qr-code-tutorial/format-version-information/#example-of-version-7-information-string

"""

if self.version < 7:

return

#Get the bit fields for this code's version

#We will iterate across the string, the bit string

#needs the least significant digit in the zero-th position

field = iter(tables.version\_pattern[self.version][::-1])

#Where to start placing the pattern

start = len(m)-11

#The version pattern is pretty odd looking

for i in range(6):

#The pattern is three modules wide

for j in range(start, start+3):

bit = int(next(field))

#Bottom Left

m[i][j] = bit

#Upper right

m[j][i] = bit

def make\_masks(self, template):

"""This method generates all seven masks so that the best mask can

be determined. The template parameter is a code matrix that will

server as the base for all the generated masks.

"""

from copy import deepcopy

nmasks = len(tables.mask\_patterns)

masks = [''] \* nmasks

count = 0

for n in range(nmasks):

cur\_mask = deepcopy(template)

masks[n] = cur\_mask

#Add the type pattern bits to the code

self.add\_type\_pattern(cur\_mask, tables.type\_bits[self.error][n])

#Get the mask pattern

pattern = tables.mask\_patterns[n]

#This will read the 1's and 0's one at a time

bits = iter(self.buffer.getvalue())

#These will help us do the up, down, up, down pattern

row\_start = itertools.cycle([len(cur\_mask)-1, 0])

row\_stop = itertools.cycle([-1,len(cur\_mask)])

direction = itertools.cycle([-1, 1])

#The data pattern is added using pairs of columns

for column in range(len(cur\_mask)-1, 0, -2):

#The vertical timing pattern is an exception to the rules,

#move the column counter over by one

if column <= 6:

column = column - 1

#This will let us fill in the pattern

#right-left, right-left, etc.

column\_pair = itertools.cycle([column, column-1])

#Go through each row in the pattern moving up, then down

for row in range(next(row\_start), next(row\_stop),

next(direction)):

#Fill in the right then left column

for i in range(2):

col = next(column\_pair)

#Go to the next column if we encounter a

#preexisting pattern (usually an alignment pattern)

if cur\_mask[row][col] != ' ':

continue

#Some versions don't have enough bits. You then fill

#in the rest of the pattern with 0's. These are

#called "remainder bits."

try:

bit = int(next(bits))

except:

bit = 0

#If the pattern is True then flip the bit

if pattern(row, col):

cur\_mask[row][col] = bit ^ 1

else:

cur\_mask[row][col] = bit

#DEBUG CODE!!!

#Save all of the masks as png files

#for i, m in enumerate(masks):

# \_png(m, self.version, 'mask-{0}.png'.format(i), 5)

return masks

def choose\_best\_mask(self):

"""This method returns the index of the "best" mask as defined by

having the lowest total penalty score. The penalty rules are defined

by the standard. The mask with the lowest total score should be the

easiest to read by optical scanners.

"""

self.scores = []

for n in range(len(self.masks)):

self.scores.append([0,0,0,0])

#Score penalty rule number 1

#Look for five consecutive squares with the same color.

#Each one found gets a penalty of 3 + 1 for every

#same color square after the first five in the row.

for (n, mask) in enumerate(self.masks):

current = mask[0][0]

counter = 0

total = 0

#Examine the mask row wise

for row in range(0,len(mask)):

counter = 0

for col in range(0,len(mask)):

bit = mask[row][col]

if bit == current:

counter += 1

else:

if counter >= 5:

total += (counter - 5) + 3

counter = 1

current = bit

if counter >= 5:

total += (counter - 5) + 3

#Examine the mask column wise

for col in range(0,len(mask)):

counter = 0

for row in range(0,len(mask)):

bit = mask[row][col]

if bit == current:

counter += 1

else:

if counter >= 5:

total += (counter - 5) + 3

counter = 1

current = bit

if counter >= 5:

total += (counter - 5) + 3

self.scores[n][0] = total

#Score penalty rule 2

#This rule will add 3 to the score for each 2x2 block of the same

#colored pixels there are.

for (n, mask) in enumerate(self.masks):

count = 0

#Don't examine the 0th and Nth row/column

for i in range(0, len(mask)-1):

for j in range(0, len(mask)-1):

if mask[i][j] == mask[i+1][j] and \

mask[i][j] == mask[i][j+1] and \

mask[i][j] == mask[i+1][j+1]:

count += 1

self.scores[n][1] = count \* 3

#Score penalty rule 3

#This rule looks for 1011101 within the mask prefixed

#and/or suffixed by four zeros.

patterns = [[0,0,0,0,1,0,1,1,1,0,1],

[1,0,1,1,1,0,1,0,0,0,0],]

#[0,0,0,0,1,0,1,1,1,0,1,0,0,0,0]]

for (n, mask) in enumerate(self.masks):

nmatches = 0

for i in range(len(mask)):

for j in range(len(mask)):

for pattern in patterns:

match = True

k = j

#Look for row matches

for p in pattern:

if k >= len(mask) or mask[i][k] != p:

match = False

break

k += 1

if match:

nmatches += 1

match = True

k = j

#Look for column matches

for p in pattern:

if k >= len(mask) or mask[k][i] != p:

match = False

break

k += 1

if match:

nmatches += 1

self.scores[n][2] = nmatches \* 40

#Score the last rule, penalty rule 4. This rule measures how close

#the pattern is to being 50% black. The further it deviates from

#this this ideal the higher the penalty.

for (n, mask) in enumerate(self.masks):

nblack = 0

for row in mask:

nblack += sum(row)

total\_pixels = len(mask)\*\*2

ratio = nblack / total\_pixels

percent = (ratio \* 100) - 50

self.scores[n][3] = int((abs(int(percent)) / 5) \* 10)

#Calculate the total for each score

totals = [0] \* len(self.scores)

for i in range(len(self.scores)):

for j in range(len(self.scores[i])):

totals[i] += self.scores[i][j]

#DEBUG CODE!!!

#Prints out a table of scores

#print('Rule Scores\n 1 2 3 4 Total')

#for i in range(len(self.scores)):

# print(i, end='')

# for s in self.scores[i]:

# print('{0: >6}'.format(s), end='')

# print('{0: >7}'.format(totals[i]))

#print('Mask Chosen: {0}'.format(totals.index(min(totals))))

#The lowest total wins

return totals.index(min(totals))

def add\_type\_pattern(self, m, type\_bits):

"""This will add the pattern to the QR code that represents the error

level and the type of mask used to make the code.

"""

field = iter(type\_bits)

for i in range(7):

bit = int(next(field))

#Skip the timing bits

if i < 6:

m[8][i] = bit

else:

m[8][i+1] = bit

if -8 < -(i+1):

m[-(i+1)][8] = bit

for i in range(-8,0):

bit = int(next(field))

m[8][i] = bit

i = -i

#Skip timing column

if i > 6:

m[i][8] = bit

else:

m[i-1][8] = bit

##############################################################################

##############################################################################

#

# Output Functions

#

##############################################################################

##############################################################################

def \_get\_writable(stream\_or\_path, mode):

"""This method returns the `stream\_or\_path` parameter if it is an open

writable stream. Otherwise it treats the `stream\_or\_path` parameter as

file path and opens it with the given mode.

It is used by the svg and png methods to interpret the file parameter.

"""

import os.path

is\_stream = hasattr(stream\_or\_path, 'write')

if not is\_stream:

# No stream provided, treat "file" as path

stream\_or\_path = open(os.path.abspath(stream\_or\_path), mode)

return stream\_or\_path, not is\_stream

def \_get\_file(file, mode):

"""This method returns a tuple containing the stream and a flag to indicate

if the stream should be automatically closed.

The file parameter is returned if it is an open writable stream. Otherwise.

it treats the file parameter as a file path and opens it with the given

mode.

It is used by the svg and png methods to interpret the file parameter.

:type file: str | io.BufferedIOBase

:type mode: str | unicode

:rtype: (io.BufferedIOBase, bool)

"""

import os.path

#See if the file parameter is a stream

if not hasattr(file, 'write'):

#If it is not a stream open a the file path

return open(os.path.abspath(file), mode), True

else:

return file, False

def \_get\_png\_size(version, scale, quiet\_zone=4):

"""See: QRCode.get\_png\_size

This function was abstracted away from QRCode to allow for the output of

QR codes during the build process, i.e. for debugging. It works

just the same except you must specify the code's version. This is needed

to calculate the PNG's size.

"""

#Formula: scale times number of modules plus the border on each side

return (int(scale) \* tables.version\_size[version]) + (2 \* quiet\_zone \* int(scale))

def \_terminal(code, module\_color='default', background='reverse', quiet\_zone=4):

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

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

QR code. The module\_color and the background color should be keys

in the tables.term\_colors table for printing using the 8/16

color scheme. Alternatively, they can be a number between 0 and

256 in order to use the 88/256 color scheme. Otherwise, a

ValueError will be raised.

Note, the code is outputted by changing the background color. Then

two spaces are written to the terminal. Finally, the terminal is

reset back to how it was.

"""

buf = io.StringIO()

def draw\_border():

for i in range(quiet\_zone):

buf.write(background)

if module\_color in tables.term\_colors:

data = '\033[{0}m \033[0m'.format(

tables.term\_colors[module\_color])

elif 0 <= module\_color <= 256:

data = '\033[48;5;{0}m \033[0m'.format(module\_color)

else:

raise ValueError('The module color, {0}, must a key in '

'pyqrcode.tables.term\_colors or a number '

'between 0 and 256.'.format(

module\_color))

if background in tables.term\_colors:

background = '\033[{0}m \033[0m'.format(

tables.term\_colors[background])

elif 0 <= background <= 256:

background = '\033[48;5;{0}m \033[0m'.format(background)

else:

raise ValueError('The background color, {0}, must a key in '

'pyqrcode.tables.term\_colors or a number '

'between 0 and 256.'.format(

background))

#This will be the beginning and ending row for the code.

border\_row = background \* (len(code[0]) + (2 \* quiet\_zone))

#Make sure we begin on a new line, and force the terminal back

#to normal

buf.write('\n')

#QRCodes have a quiet zone consisting of background modules

for i in range(quiet\_zone):

buf.write(border\_row)

buf.write('\n')

for row in code:

#Each code has a quiet zone on the left side, this is the left

#border for this code

draw\_border()

for bit in row:

if bit == 1:

buf.write(data)

elif bit == 0:

buf.write(background)

#Each row ends with a quiet zone on the right side, this is the

#right hand border background modules

draw\_border()

buf.write('\n')

#QRCodes have a background quiet zone row following the code

for i in range(quiet\_zone):

buf.write(border\_row)

buf.write('\n')

return buf.getvalue()

def \_text(code, quiet\_zone=4):

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

This is useful for debugging purposes.

"""

buf = io.StringIO()

border\_row = '0' \* (len(code[0]) + (quiet\_zone\*2))

#Every QR code start with a quiet zone at the top

for b in range(quiet\_zone):

buf.write(border\_row)

buf.write('\n')

for row in code:

#Draw the starting quiet zone

for b in range(quiet\_zone):

buf.write('0')

#Actually draw the QR code

for bit in row:

if bit == 1:

buf.write('1')

elif bit == 0:

buf.write('0')

#This is for debugging unfinished QR codes,

#unset pixels will be spaces.

else:

buf.write(' ')

#Draw the ending quiet zone

for b in range(quiet\_zone):

buf.write('0')

buf.write('\n')

#Every QR code ends with a quiet zone at the bottom

for b in range(quiet\_zone):

buf.write(border\_row)

buf.write('\n')

return buf.getvalue()

def \_svg(code, version, 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 function 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 (binary)

stream or a file path. The scale parameter is sets how large to draw

a single module. By default one pixel 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).

:param module\_color: Color of the QR code (default: ``#000`` (black))

:param background: Optional background color.

(default: ``None`` (no background))

:param quiet\_zone: Border around the QR code (also known as quiet zone)

(default: ``4``). Set to zero (``0``) if the code shouldn't

have a border.

:param xmldecl: Inidcates if the XML declaration header should be written

(default: ``True``)

:param svgns: Indicates if the SVG namespace should be written

(default: ``True``)

:param title: Optional title of the generated SVG document.

:param svgclass: The CSS class of the SVG document

(if set to ``None``, the SVG element won't have a class).

:param lineclass: The CSS class of the path element

(if set to ``None``, the path won't have a class).

:param omithw: Indicates if width and height attributes should be

omitted (default: ``False``). If these attributes are omitted,

a ``viewBox`` attribute will be added to the document.

:param debug: Inidicates if errors in the QR code should be added to the

output (default: ``False``).

"""

from functools import partial

from xml.sax.saxutils import quoteattr

def write\_unicode(write\_meth, unicode\_str):

"""\

Encodes the provided string into UTF-8 and writes the result using

the `write\_meth`.

"""

write\_meth(unicode\_str.encode('utf-8'))

def line(x, y, length, relative):

"""Returns coordinates to draw a line with the provided length.

"""

return '{0}{1} {2}h{3}'.format(('m' if relative else 'M'), x, y, length)

def errline(col\_number, row\_number):

"""Returns the coordinates to draw an error bit.

"""

# Debug path uses always absolute coordinates

# .5 == stroke / 2

return line(col\_number + quiet\_zone, row\_number + quiet\_zone + .5, 1, False)

f, autoclose = \_get\_writable(file, 'wb')

write = partial(write\_unicode, f.write)

write\_bytes = f.write

# Write the document header

if xmldecl:

write\_bytes(b'<?xml version="1.0" encoding="UTF-8"?>\n')

write\_bytes(b'<svg')

if svgns:

write\_bytes(b' xmlns="http://www.w3.org/2000/svg"')

size = tables.version\_size[version] \* scale + (2 \* quiet\_zone \* scale)

if not omithw:

write(' height="{0}" width="{0}"'.format(size))

else:

write(' viewBox="0 0 {0} {0}"'.format(size))

if svgclass is not None:

write\_bytes(b' class=')

write(quoteattr(svgclass))

write\_bytes(b'>')

if title is not None:

write('<title>{0}</title>'.format(title))

# Draw a background rectangle if necessary

if background is not None:

write('<path fill="{1}" d="M0 0h{0}v{0}h-{0}z"/>'

.format(size, background))

write\_bytes(b'<path')

if scale != 1:

write(' transform="scale({0})"'.format(scale))

if module\_color is not None:

write\_bytes(b' stroke=')

write(quoteattr(module\_color))

if lineclass is not None:

write\_bytes(b' class=')

write(quoteattr(lineclass))

write\_bytes(b' d="')

# Used to keep track of unknown/error coordinates.

debug\_path = ''

# Current pen pointer position

x, y = -quiet\_zone, quiet\_zone - .5 # .5 == stroke-width / 2

wrote\_bit = False

# Loop through each row of the code

for rnumber, row in enumerate(code):

start\_column = 0 # Reset the starting column number

coord = '' # Reset row coordinates

y += 1 # Pen position on y-axis

length = 0 # Reset line length

# Examine every bit in the row

for colnumber, bit in enumerate(row):

if bit == 1:

length += 1

else:

if length:

x = start\_column - x

coord += line(x, y, length, relative=wrote\_bit)

x = start\_column + length

y = 0 # y-axis won't change unless the row changes

length = 0

wrote\_bit = True

start\_column = colnumber + 1

if debug and bit != 0:

debug\_path += errline(colnumber, rnumber)

if length:

x = start\_column - x

coord += line(x, y, length, relative=wrote\_bit)

x = start\_column + length

wrote\_bit = True

write(coord)

# Close path

write\_bytes(b'"/>')

if debug and debug\_path:

write\_bytes(b'<path')

if scale != 1:

write(' transform="scale({0})"'.format(scale))

write(' class="pyqrerr" stroke="red" d="{0}"/>'.format(debug\_path))

# Close document

write\_bytes(b'</svg>\n')

if autoclose:

f.close()

def \_png(code, version, file, scale=1, module\_color=(0, 0, 0, 255),

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

"""See: pyqrcode.QRCode.png()

This function was abstracted away from QRCode to allow for the output of

QR codes during the build process, i.e. for debugging. It works

just the same except you must specify the code's version. This is needed

to calculate the PNG's size.

This method will write the given file out as a PNG file. Note, it

depends on the PyPNG module to do this.

:param module\_color: Color of the QR code (default: ``(0, 0, 0, 255)`` (black))

:param background: Optional background color. If set to ``None`` the PNG

will have a transparent background.

(default: ``(255, 255, 255, 255)`` (white))

:param quiet\_zone: Border around the QR code (also known as quiet zone)

(default: ``4``). Set to zero (``0``) if the code shouldn't

have a border.

:param debug: Inidicates if errors in the QR code should be added (as red

modules) to the output (default: ``False``).

"""

try:

import png

except ImportError:

from . import png

# Coerce scale parameter into an integer

try:

scale = int(scale)

except ValueError:

raise ValueError('The scale parameter must be an integer')

def scale\_code(size):

"""To perform the scaling we need to inflate the number of bits.

The PNG library expects all of the bits when it draws the PNG.

Effectively, we double, tripple, etc. the number of columns and

the number of rows.

"""

# This is one row's worth of each possible module

# PNG's use 0 for black and 1 for white, this is the

# reverse of the QR standard

black = [0] \* scale

white = [1] \* scale

# Tuple to lookup colors

# The 3rd color is the module\_color unless "debug" is enabled

colors = (white, black, (([2] \* scale) if debug else black))

# Whitespace added on the left and right side

border\_module = white \* quiet\_zone

# This is the row to show up at the top and bottom border

border\_row = [[1] \* size] \* scale \* quiet\_zone

# This will hold the final PNG's bits

bits = []

# Add scale rows before the code as a border,

# as per the standard

bits.extend(border\_row)

# Add each row of the to the final PNG bits

for row in code:

tmp\_row = []

# Add one all white module to the beginning

# to create the vertical border

tmp\_row.extend(border\_module)

# Go through each bit in the code

for bit in row:

# Use the standard color or the "debug" color

tmp\_row.extend(colors[(bit if bit in (0, 1) else 2)])

# Add one all white module to the end

# to create the vertical border

tmp\_row.extend(border\_module)

# Copy each row scale times

for n in range(scale):

bits.append(tmp\_row)

# Add the bottom border

bits.extend(border\_row)

return bits

def png\_pallete\_color(color):

"""This creates a palette color from a list or tuple. The list or

tuple must be of length 3 (for rgb) or 4 (for rgba). The values

must be between 0 and 255. Note rgb colors will be given an added

alpha component set to 255.

The pallete color is represented as a list, this is what is returned.

"""

if color is None:

return ()

if not isinstance(color, (tuple, list)):

r, g, b = \_hex\_to\_rgb(color)

return r, g, b, 255

rgba = []

if not (3 <= len(color) <= 4):

raise ValueError('Colors must be a list or tuple of length '

' 3 or 4. You passed in "{0}".'.format(color))

for c in color:

c = int(c)

if 0 <= c <= 255:

rgba.append(int(c))

else:

raise ValueError('Color components must be between 0 and 255')

# Make all colors have an alpha channel

if len(rgba) == 3:

rgba.append(255)

return tuple(rgba)

if module\_color is None:

raise ValueError('The module\_color must not be None')

bitdepth = 1

# foreground aka module color

fg\_col = png\_pallete\_color(module\_color)

transparent = background is None

# If background color is set to None, the inverse color of the

# foreground color is calculated

bg\_col = png\_pallete\_color(background) if background is not None else tuple([255 - c for c in fg\_col])

# Assume greyscale if module color is black and background color is white

greyscale = fg\_col[:3] == (0, 0, 0) and (not debug and transparent or bg\_col == (255, 255, 255, 255))

transparent\_color = 1 if transparent and greyscale else None

palette = [fg\_col, bg\_col] if not greyscale else None

if debug:

# Add "red" as color for error modules

palette.append((255, 0, 0, 255))

bitdepth = 2

# The size of the PNG

size = \_get\_png\_size(version, scale, quiet\_zone)

# We need to increase the size of the code to match up to the

# scale parameter.

code\_rows = scale\_code(size)

# Write out the PNG

f, autoclose = \_get\_writable(file, 'wb')

w = png.Writer(width=size, height=size, greyscale=greyscale,

transparent=transparent\_color, palette=palette,

bitdepth=bitdepth)

try:

w.write(f, code\_rows)

finally:

if autoclose:

f.close()

def \_eps(code, version, file\_or\_path, scale=1, module\_color=(0, 0, 0),

background=None, quiet\_zone=4):

"""This function writes the QR code out as an EPS 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 (text)

stream or a file path. The scale parameter is 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 function will accept

fractional scales (e.g. 2.5).

:param module\_color: Color of the QR code (default: ``(0, 0, 0)`` (black))

The color can be specified as triple of floats (range: 0 .. 1) or

triple of integers (range: 0 .. 255) or as hexadecimal value (i.e.

``#36c`` or ``#33B200``).

:param background: Optional background color.

(default: ``None`` (no background)). See `module\_color` for the

supported values.

:param quiet\_zone: Border around the QR code (also known as quiet zone)

(default: ``4``). Set to zero (``0``) if the code shouldn't

have a border.

"""

from functools import partial

import time

def write\_line(writemeth, content):

"""\

Writes `content` and ``LF``.

"""

writemeth(content + '\n')

def line(offset, length):

"""\

Returns coordinates to draw a line with the provided length.

"""

res = ''

if offset > 0:

res = ' {0} 0 m'.format(offset)

res += ' {0} 0 l'.format(length)

return res

def rgb\_to\_floats(color):

"""\

Converts the provided color into an acceptable format for Postscript's

``setrgbcolor``

"""

def to\_float(clr):

if isinstance(clr, float):

if not 0.0 <= clr <= 1.0:

raise ValueError('Invalid color "{0}". Not in range 0 .. 1'

.format(clr))

return clr

if not 0 <= clr <= 255:

raise ValueError('Invalid color "{0}". Not in range 0 .. 255'

.format(clr))

return 1/255.0 \* clr if clr != 1 else clr

if not isinstance(color, (tuple, list)):

color = \_hex\_to\_rgb(color)

return tuple([to\_float(i) for i in color])

f, autoclose = \_get\_writable(file\_or\_path, 'w')

writeline = partial(write\_line, f.write)

size = tables.version\_size[version] \* scale + (2 \* quiet\_zone \* scale)

# Write common header

writeline('%!PS-Adobe-3.0 EPSF-3.0')

writeline('%%Creator: PyQRCode <https://pypi.python.org/pypi/PyQRCode/>')

writeline('%%CreationDate: {0}'.format(time.strftime("%Y-%m-%d %H:%M:%S")))

writeline('%%DocumentData: Clean7Bit')

writeline('%%BoundingBox: 0 0 {0} {0}'.format(size))

# Write the shortcuts

writeline('/M { moveto } bind def')

writeline('/m { rmoveto } bind def')

writeline('/l { rlineto } bind def')

mod\_color = (0, 0, 0) if module\_color == (0, 0, 0) else rgb\_to\_floats(module\_color)

if background is not None:

writeline('{0:f} {1:f} {2:f} setrgbcolor clippath fill'

.format(\*rgb\_to\_floats(background)))

if mod\_color == (0, 0, 0):

# Reset RGB color back to black iff module color is black

# In case module color != black set the module RGB color later

writeline('0 0 0 setrgbcolor')

if mod\_color != (0, 0, 0):

writeline('{0:f} {1:f} {2:f} setrgbcolor'.format(\*rgb\_to\_floats(module\_color)))

if scale != 1:

writeline('{0} {0} scale'.format(scale))

writeline('newpath')

# Current pen position y-axis

# Note: 0, 0 = lower left corner in PS coordinate system

y = tables.version\_size[version] + quiet\_zone + .5 # .5 = linewidth / 2

last\_bit = 1

# Loop through each row of the code

for row in code:

offset = 0 # Set x-offset of the pen

length = 0

y -= 1 # Move pen along y-axis

coord = '{0} {1} M'.format(quiet\_zone, y) # Move pen to initial pos

for bit in row:

if bit != last\_bit:

if length:

coord += line(offset, length)

offset = 0

length = 0

last\_bit = bit

if bit == 1:

length += 1

else:

offset += 1

if length:

coord += line(offset, length)

writeline(coord)

writeline('stroke')

writeline('%%EOF')

if autoclose:

f.close()

def \_hex\_to\_rgb(color):

"""\

Helper function to convert a color provided in hexadecimal format

as RGB triple.

"""

if color[0] == '#':

color = color[1:]

if len(color) == 3:

color = color[0] \* 2 + color[1] \* 2 + color[2] \* 2

if len(color) != 6:

raise ValueError('Input #{0} is not in #RRGGBB format'.format(color))

return [int(n, 16) for n in (color[:2], color[2:4], color[4:])]