PyQRCodeNG Documentation

Author

Jul 07, 2019

Contents

The PyQRCodeNG module is a QR code generator that is simple to use and written in pure python. The module is compatible with Python 2.6, 2.7, and 3.x. The module automates most of the building process for you. Generally, QR codes can be created using only two lines of code!

Unlike many other generators, all of the automation can be controlled manually. You are free to set any or all of the properties of your QR code.

QR codes can be saved as SVG, EPS, PNG (by using the pypng module), and plain text. PIL is not used to render the image files. You can also display a QR code directly in a compatible terminal.

The PyQRCodeNG module attempts to follow the QR code standard as closely as possible. The terminology and the encodings used in pyqrcode come directly from the standard. This module also follows the algorithm laid out in the standard.

Contents:

CHAPTER 1

QR Code creation from the command line

The command line script "pyqr" can be used to print QR Codes to the command line or to serialize QR Codes.

1.1 Usage

Output the QR Code to the terminal:

```
$ pyqr "Little wing"
```

1.1.1 Version

If the `version` parameter is not provided, pyqr chooses the minimal version for the QR Code automatically. The version may be specified as an integer.

The content 'Layla' would fit into a version 1 QR Code, but the following commands enforce version 5:

```
$ pyqr --version=5 Layla
$ pyqr -v=5 Layla
```

1.1.2 Error correction level

The default error correction level is "H", use the `error` parameter to change it:

```
$ pyqr --error=q "Ain't no grave"
$ pyqr -e=m "Heart of Gold"
```

1.1.3 QR Code serialization

Printing the QR Codes to the terminal is nice but the `output` parameter serializes the QR Code in one of the supported file formats:

```
$ pyqr --output=white-room.png "White Room"
$ pyqr -o=satellite.svg "Satellite Of Love"
$ pyqr --output=mrs.eps "Mrs. Robinson"
```

1.1.4 Scaling QR Codes

If the resulting QR Code is too small, `scale` can be used to create a more appropriate output:

```
$ pyqr --scale=10 --output=money-talks.png "Money Talks"
$ pyqr -s 10 --output=private-investigations.svg Private Investigations
```

If the serializer does not support a scaling factor (i.e. text output), this parameter is ignored.

1.1.5 Changing the size of the quiet zone

The generated QR Codes will have a recommended quiet zone around the symbol. To change the size of the quiet zone, `quietzone` can be utilized:

```
$ pyqr --quietzone=0 --output=black-magic-woman.svg "Black Magic Woman"
$ pyqr --qz=10 --output=diamond.png "Shine On You Crazy Diamond"
```

CHAPTER 2

Creating QR Codes

The QRCode object is designed to be smart about how it constructs QR codes. It can automatically figure out what mode and version to use to construct a QR code, based on the data and the amount error correction. The error correction level defaults to the highest possible level of error correction.

Below are some examples of creating QR Codes using the automatated system.

```
>>> url = pyqrcodeng.create('http://uca.edu')
>>> url = pyqrcodeng.create('http://uca.edu', error='L')
```

There are many situations where you might wish to have more fine grained control over how the QR Code is generated. You can specify all the properties of your QR code through the optional parameters of the *pyqrcodeng.create()* function. There are three main properties to a QR code.

The *error* parameter sets the error correction level of the code. Each level has an associated name given by a letter: L, M, Q, or H; each level can correct up to 7, 15, 25, or 30 percent of the data respectively. There are several ways to specify the level, see `pyqrcodeng.tables.error_level` 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 for a given version.

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. By default, the object uses the data's encoding and error correction level to calculate the smallest possible version. 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.

Finally, the *mode* parameter sets how the contents will be encoded. Three of the four possible encodings are available. By default, the object uses the most efficient encoding for the contents. You can override this behavior by setting this parameter. See `pyqrcodeng.tables.modes` for a list of possible values for this parameter. A much longer discussion on modes can be found in the next section *Encoding Data*.

The code below constructs a QR code with 25% error correction, size 27, and forces the encoding to be binary (rather than numeric).

```
>>> big_code = pyqrcodeng.create('0987654321', error='L', version=27, mode='binary')
```

CHAPTER 3

Encoding Data

The standard calls the data's encoding its *mode*. The QR code standard defines how to encode any given piece of data. There are four possible modes. This module supports three of them: numeric, alphanumeric, and binary.

Each mode is worse at encoding the QR code's contents. In other words, each mode will require more room in the QR code to store the data. How much data a code version can hold is dependent on what mode is used and the error correction level. For example, the binary encoding always requires more code words than the numeric encoding.

Because of this, it is *generally* better to allow the QRCode object to auto-select the most efficient mode for the code's contents.

Note: The QRCode object can automatically choose the best mode based on the data to be encoded. In general, it is best to just let the object figure it out for you.

3.1 Numeric Encoding

The numeric type is the most efficient way to encode digits. Problematically, the standard make no provisions for encoding negative or fractional numbers. This encoding is better than Alphanumeric, when you only have a list of digits.

To use this encoding, simply specify a string of digits as the data. You can also use a positive integer as the code's contents.

```
>>> number = pyqrcodeng.create(123456789012345)
 >>> number2 = pyqrcodeng.create('0987654321')
```

3.2 Alphanumeric

The alphanumeric type is very limited in that it can only encode some ASCII characters. It encodes:

- Uppercase letters
- Digits 0-9
- The horizontal space
- Eight punctuation characters: $, %, *, +, -, ., /, and :

A complete list of the possible characters can be found in the `pyqrcodeng.tables.ascii_codes` dictionary. While limited, this encoding is much more efficient than using the binary encoding, in many cases. Luckily, the available characters will let you encode a URL.

```
>>> url = pyqrcodeng.create('http://uca.edu'.upper())
```

3.3 Kanji

The final mode allows for the encoding of Kanji characters. Denso Wave, the creators of the QR code, is a Japenese company. Hence, they made special provisions for using QR codes with Japenese text.

Only one python string encoding for Kanji characters is supported, shift-jis. The auto-detection algorithm will try to encode the given string as shift-jis. if the characters are supported, then the mode will be set to kanji. Alternatively, you can explicitly define the data's encoding.

```
>>> utf8 = ''.encode('utf-8')
>>> monty = pyqrcodeng.create(utf8, encoding='utf-8')
>>> python = pyqrcodeng.create('')
```

3.4 Binary

When all else fails the data can be encoded in pure binary. This encoding does not change the data in any way. Instead its pure bytes are represented directly in the QR code. This is the least efficient way to store data in a QR code. You should only use this as a last resort.

The quotation below must be encoded in binary because of the apostrophe, exclamation point, and the new line character. Notice, that the string's characters will not have their case changed.

```
>>> life = pyqrcodeng.create('''MR. CREOSOTE: Better get a bucket. I'm going to throw␣
↪up.
   MAITRE D: Uh, Gaston! A bucket for monsieur. There you are, monsieur.''')
```

CHAPTER 4

Rendering QR Codes

There are five possible formats for rendering the QR Code. The first is to render it as a string of 1's and 0's. Next, the code can be displayed directly in compatible terminals. There are also three image based renderers. All, but the first, allow you to set the colors used. They also take a scaling factor, that way each module is not rendered as 1 pixel.

4.1 Text Based Rendering

The PyQRCodeNG module includes a basic text renderer. This will return a string containing the QR code as a string of 1's and 0's, with each row of the code on a new line. A *data module* in the QR Code is represented by a 1. Likewise, 0 is used to represent the background of the code.

The purpose of this renderer is to allow users to create their own renderer if none of the built in renderers are satisfactory.

```
>>> number = pyqrcodeng.create(123)
>>> print(number.text())
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00001111111011110011111110000
00001000001000101010000010000
00001011101001010010111010000
00001011101010011010111010000
00001011101000100010111010000
00001000001001001010000010000
00001111111010101011111110000
00000000000001011000000000000
00000101110110101000100010000
00001011110001111101010010000
00000111111011100101001000000
00001001100011010011110010000
00001111111001101011001110000
```

(continues on next page)

(continued from previous page)

```
000000000000010000000001100000
000011111110001111001001000000
000010000010110101100011000000
000010111010101100001010100000
000010111010011111110101000000
000010111010111010011010000000
000010000010010110011100000000
000011111110000110110110100000
000000000000000000000000000000
000000000000000000000000000000
000000000000000000000000000000
000000000000000000000000000000
```

4.2 Terminal Rendering

QR codes can be directly rendered to a compatible terminal in a manner readable by QR code scanners. The rendering is done using ASCII escape codes. Hence, most Linux terminals are supported.

```
>>> text = pyqrcodeng.create('Example')
>>> text.term()
```

4.3 Image Rendering

There are four ways to get an image of the generated QR code. All of the renderers have a few things in common.

Each renderer takes a file path or writable stream and draws the QR code there. The methods should auto-detect which is which.

Each renderer takes a scale parameter. This parameter sets the size of a single *data module* in pixels. Setting this parameter to one, will result in each *data module* taking up 1 pixel. In other words, the QR code would be too small to scan. What scale to use depends on how you plan to use the QR code. Generally, three, four, or five will result in small but scanable QR codes.

QR codes are also supposed to have a *quiet zone* around them. This area is four modules wide on each side. The purpose of the quiet zone is to make scanning a printed area more reliable. For electronic usages, this may be unnecessary depending on how the code is being displayed. Each of the renderers allows you to set the size of the quiet zone.

Many of the renderers, also, allow you to set the *module* and background colors. Although, how the colors are represented are renderer specific.

4.3.1 XBM Rendering

The XBM file format is a simple black and white image format. The image data takes the form of a valid C header file. XBM rendering is handled via the *pyqrcodeng.QRCode.xbm()* method.

XMB's are natively supported by Tkinter. This makes displaying QR codes in a Tkinter application very simple.

```
>>> import pyqrcodeng
>>> import tkinter
>>> # Create and render the QR code
>>> code = pyqrcodeng.create('Knights who say ni!')
```

(continues on next page)

(continued from previous page)

```
>>> code_xbm = code.xbm(scale=5)
>>> # Create a tk window
>>> top = tkinter.Tk()
>>> # Make generate the bitmap image from the redered code
>>> code_bmp = tkinter.BitmapImage(data=code_xbm)
>>> # Set the code to have a white background,
>>> # instead of transparent
>>> code_bmp.config(background="white")
>>> # Bitmaps are accepted by lots of Widgets
>>> label = tkinter.Label(image=code_bmp)
>>> # The QR code is now visible
>>> label.pack()
```

4.3.2 Scalable Vector Graphic (SVG)

The SVG renderer outputs the QR code as a scalable vector graphic using the *pyqrcodeng.QRCode.svg()* method.

The method draws the QR code using a set of paths. By default, no background is drawn, i.e. the resulting code has a transparent background. The default foreground (module) color is black.

```
>>> url = pyqrcodeng.create('http://uca.edu')
>>> url.svg('uca.svg', scale=4)
>>> # in-memory stream is also supported
>>> buffer = io.BytesIO()
>>> url.svg(buffer)
>>> # do whatever you want with buffer.getvalue()
>>> print(list(buffer.getvalue()))
```

You can change the colors of the data-modules using the *module_color* parameter. Likewise, you can specify a background using the *background* parameter. Each of these parameters take a HTML style color.

```
>>> url.svg('uca.svg', scale=4, background="white", module_color="#7D007D")
```

You can also suppress certain parts of the SVG document. In other words you can create a SVG fragment.

4.3.3 Encapsulated PostScript (EPS)

The EPS renderer outputs the QR code an encapsulated PostScript document using the *pyqrcodeng.QRCode.eps()* method. *This renderer does not require any external modules.*

The method draws the EPS document using lines of contiguous modules. By default, no background is drawn, i.e. the resulting code has a transparent background. The default module color is black. Note, that a scale of 1 equates to a module being drawn at 1 point (1/72 of an inch).

```
>>> qr = pyqrcodeng.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))
```

4.3.4 Portable Network Graphic (PNG)

The PNG renderer outputs the QR code as a portable network graphic file using the *pyqrcodeng.QRCode.png()* method.

Note: This renderer requires the PyPNG module.

```
>>> url = pyqrcodeng.create('http://uca.edu')
>>> with open('code.png', 'w') as fstream:
...     url.png(fstream, scale=5)
>>> # same as above
>>> url.png('code.png', scale=5)
>>> # in-memory stream is also supported
>>> buffer = io.BytesIO()
>>> url.png(buffer)
>>> # do whatever you want with buffer.getvalue()
>>> print(list(buffer.getvalue()))
```

Colors should be a list or tuple containing numbers between zero an 255. The lists should be of length three (for RGB) or four (for RGBA). The color (0,0,0) represents black and the color (255,255,255) represents white. A value of zero for the fourth element, represents full transparency. Likewise, a value of 255 for the fourth element represents full opacity.

By default, the renderer creates a QR code with the data modules colored black, and the background modules colored white.

```
>>> url.png('uca-colors.png', scale=6,
...         module_color=[0, 0, 0, 128],
...         background=[0xff, 0xff, 0xcc])
```

CHAPTER 5

PyQRCodeNG Module Documentation

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 *pyqrcodeng.create()* function to build the *QRCode* object. This results in cleaner looking code.

Examples:

```
>>> import pyqrcodeng
>>> import sys
>>> url = pyqrcodeng.create('http://uca.edu')
>>> url.svg(sys.stdout, scale=1)
>>> url.svg('uca.svg', scale=4)
>>> number = pyqrcodeng.create(123456789012345)
>>> number.png('big-number.png')
```

pyqrcodeng.**create**(*content*, *error='H'*, *version=None*, *mode=None*, *encoding=None*)

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 *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 mode for the contents is guessed. There are four possible modes. 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. The is 'kanji' mode can be used for Japanese characters, but only those that can be understood via the shift-jis string encoding. Finally, we then have 'binary' mode which just encodes the bytes directly into the QR code (this encoding is the least efficient).

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 or None. It will be passed the *content*'s encode/decode methods.

class pyqrcodeng.**QRCode**(*content*, *error='H'*, *version=None*, *mode=None*, *encoding=None*)

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 pyqrcodeng 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 *pyqrcodeng.create()* function.

eps(*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 = pyqrcodeng.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))
```

get_png_size(*scale=1*, *quiet_zone=4*)

DEPRECATED, use *pyqrcodeng.QRCode.symbol_size()*

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.

Deprecated since version 1.3.0.

Example:

```
>>> code = pyqrcodeng.QRCode("I don't like spam!")
>>> print(code.symbol_size(1))
(31, 31)
>>> print(code.symbol_size(5))
(155, 155)
```

png(*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.

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). You can use the *symbol_size()* method to calculate the actual pixel size of the resulting PNG image.

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 = pyqrcodeng.create('Are you suggesting coconuts migrate?')
>>> code.png('swallow.png', scale=5)
```

(continues on next page)

(continued from previous page)

```
>>> code.png('swallow.png', scale=5,
             module_color=(0x66, 0x33, 0x0),      # Dark brown
             background=(0xff, 0xff, 0xff, 0x88)) # 50% transparent white
```

png_as_base64_str(*scale=1*, *module_color=(0, 0, 0, 255)*, *background=(255, 255, 255, 255)*, *quiet_zone=4*)

DEPRECATED, use *pyqrcodeng.QRCode.png_data_uri()*.

This method uses the png render and returns the PNG image encoded as base64 string. This can be useful for creating dynamic PNG images for web development, since no file needs to be created.

Example:

```
>>> code = pyqrcodeng.create('Are you suggesting coconuts migrate?')
>>> image_as_str = code.png_as_base64_str(scale=5)
>>> html_img = '<img src="{}">'.format(image_as_str)
```

The parameters are passed directly to the *png()* method. Refer to that method's documentation for the meaning behind the parameters.

Deprecated since version 1.3.0.

Note: This method depends on the Segno package to actually create the PNG image.

---

png_data_uri(*scale=1*, *module_color=(0, 0, 0, 255)*, *background=(255, 255, 255, 255)*, *quiet_zone=4*)

Converts the QR Code into a PNG data URI.

Uses the same keyword parameters as the usual PNG serializer, see *QRCode.png()* for details.

Return type str

show(*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.

svg(*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 = pyqrcodeng.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')
```

symbol_size(*scale=1*, *quiet_zone=4*)

Returns the symbol size (width x height) with the provided border and scaling factor.

Parameters

- **scale** (*int or float*) – Indicates the size of a single module (default: 1). The size of a module depends on the used output format; i.e. in a PNG context, a scaling factor of 2 indicates that a module has a size of 2 x 2 pixel. Some outputs (i.e. SVG) accept floating point values.
- **quiet_zone** (*int*) – The size of the quiet zone.

Return type tuple (width, height)

term(*file=None*, *quiet_zone=4*)

This method prints the QR code to the terminal.

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. If `file` is `None` (default) the code is written to `sys.stdout`.

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 = pyqrcodeng.create('Example')
>>> code.term()
```

terminal(*module_color='default'*, *background='reverse'*, *quiet_zone=4*)
DEPRECATED, use *term()*

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.

Deprecated since version 1.3.0.

Example:

```
>>> code = pyqrcodeng.create('Example')
>>> text = code.terminal()
>>> print(text)
```

text(*scale=1*, *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 act a starting point for users to create their own renderers.

The *scale* parameter sets how large to draw a single module. By default one value ("0" for a light module or "1" for a dark module) is used to draw a single module.

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 = pyqrcodeng.create('Example')
>>> text = code.text()
>>> print(text)
```

xbm(*scale=1*, *quiet_zone=4*)
Returns a string representing an XBM image of the QR code. The XBM format is a black and white image format that looks like a C header file.

Because displaying QR codes in Tkinter is the primary use case for this renderer, this method does not take a file parameter. Instead it retuns the rendered QR code data as a string.

Example of using this renderer with Tkinter:

```
>>> import pyqrcodeng
>>> import tkinter
>>> code = pyqrcodeng.create('Knights who say ni!')
>>> code_xbm = code.xbm(scale=5)
>>>
>>> top = tkinter.Tk()
>>> code_bmp = tkinter.BitmapImage(data=code_xbm)
>>> code_bmp.config(foreground="black")
>>> code_bmp.config(background="white")
>>> label = tkinter.Label(image=code_bmp)
>>> label.pack()
```

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). You can use the *symbol_size()* method to calculate the actual pixel size of this image when displayed.

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.

CHAPTER 6

Glossary

error

error level QR codes can use one of four possible error correction values. They are referred to by the letters: L, M, Q, and H. The *L* error correction level corresponds to 7% of the code can be corrected. The *M* error correction level corresponds to 15% of the code can be corrected. The *Q* error correction level corresponds to 25% of the code can be corrected. The *H* error correction level corresponds to 30% of the code can be corrected.

mode The encoding used to represent the data in a QR code. There are four possible encodings: binary, numeric, alphanumeric, kanji.

module

data module A square dot on a QR code. Generally, only the "black" dots count. The "white" squares are considered part of the background.

quiet zone An empty area around the QR code. The area is the background module in color. According to the standard this area should be four modules wide.

QR code

Quick Response code A two dimensional barcode developed by Denso Wave.

version A version is one of 40 different possible sizes a QR code comes in. The version of a QR Code determines it maximum possible data capacity.

CHAPTER 7

Requirements

The PyQRCodeNG module only requires Python 2.6, 2.7, 3.x. You may want to install pypng in order to render PNG files, but it is optional. Note, pypng is a pure python PNG writer which does not require any other libraries.

CHAPTER 8

Installation

Installation is simple. PyQRCodeNG can be installed from pip using the following command:

```
$ pip install -U pyqrcodeng
```

CHAPTER 9

Replacing PyQRCode with PyQRCodeNG

PyQRCodeNG is a fork of PyQRCode since the latter seems to be unmaintained. The API is mainly compatible to PyQRCode. In your code you can use the following import without changing the QR Code generation code:

```
>>> import pyqrcodeng as pyqrcode
```

CHAPTER 10

Usage

The PyQRCodeNG module aims to be as simple to use as possible. Below is a simple example of creating a QR code for a URL. The code is rendered out as a black and white scalable vector graphics file.

```
>>> import pyqrcodeng
>>> url = pyqrcodeng.create('http://uca.edu')
>>> url.svg('uca-url.svg', scale=8)
>>> print(url.terminal(quiet_zone=1))
```

The PyQRCodeNG module, while easy to use, is powerful. You can set all of the properties of the QR code. If you install the optional pypng library, you can also render the code as a PNG image. Below is a more complex example:

```
>>> big_code = pyqrcodeng.create('0987654321', error='L', version=27, mode='binary')
>>> big_code.png('code.png', scale=6, module_color=[0, 0, 0, 128], background=[0xff,␣
↪0xff, 0xcc])
```

CHAPTER 11

Developer Documentation

11.1 Special QR Codes

Generation of special-purpose text for Qr codes.

class pyqrcodeng.qrspecial.**QrGeolocation**(*lat=None*, *lon=None*, *query=None*)
QrSpecial-derived geolocation.

classmethod from_str(*text*, *strict=True*, *strip=True*)
Construct a QrSpecial object from its QR-ready text.

This is conceptually the inverse operation of the 'to_str' method.

Args: text (str|unicode): The input text. strict (bool): Raises an error if tags are missing. strip (bool): Strip from whitespaces before parsing.

Returns: obj (QrSpecial): The QrSpecial object.

class pyqrcodeng.qrspecial.**QrMeCard**(*name=None*, *reading=None*, *tel=None*, *telav=None*, *email=None*, *memo=None*, *birthday=None*, *address=None*, *url=None*, *nickname=None*, *company=None*)
QrSpecial-derived contact information (MeCard).

class pyqrcodeng.qrspecial.**QrShortMessage**(*number=None*, *text=None*)
QrSpecial-derived short message (SMS).

class pyqrcodeng.qrspecial.**QrSpecial**(***kws*)
Special-purpose text for QR codes.

Implements the special text generated by the ZXing project for QR codes. Likely, these are correctly handled by software using the this library.

Of note:

- the *Event* special text is not supported here, but it can be handled by using the *icalendar* package [https://pypi.python.org/pypi/icalendar].

- the *vCard* contact format is not supported here (only MeCard), but a number of packages for handling vCards are available in PyPI.

classmethod from_str(*text*, *strict=True*, *strip=True*)
Construct a QrSpecial object from its QR-ready text.

This is conceptually the inverse operation of the 'to_str' method.

Args: text (str|unicode): The input text. strict (bool): Raises an error if tags are missing. strip (bool): Strip from whitespaces before parsing.

Returns: obj (QrSpecial): The QrSpecial object.

static parse(*text*)
Construct a QrSpecial-derived object from a text.

This can be useful for determining whether a given input is a valid QrSpecial-derived object.

Args: text (str|unicode): The input text.

Returns: obj (QrSpecial): The QrSpecial-derived object.

class pyqrcodeng.qrspecial.**QrWifi**(*ssid=None*, *security=None*, *password=None*, *hidden=None*)
QrSpecial-derived WiFi network.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index