PyQRCode 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 `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')
```

pyqrcode.create(*content, error='H', version=None, mode=None, encoding=None*) [source]

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. t will be passed the *content*'s encode/decode methods.

class **pyqrcode.QRCode**(*content*, *error='H'*, *version=None*, *mode=None*, *encoding='iso-8859-1'*) [source]

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 `pyqrcode.create()` function.

eps(*file*, *scale=1*, *module_color=(0, 0, 0)*, *background=None*, *quiet_zone=4*) [source]

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))
```

get_png_size(*scale=1*, *quiet_zone=4*) [source]

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
```

png(*file*, *scale=1*, *module_color=(0, 0, 0, 255)*, *background=(255, 255, 255, 255)*, *quiet_zone=4*) [source]

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). You can use the `get_png_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 = 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
```

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

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 = pyqrcode.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.

Note

This method depends on the pypng module to actually create the PNG image.

show(*wait=1.2*, *scale=10*, *module_color=(0, 0, 0, 255)*, *background=(255, 255, 255, 255)*, *quiet_zone=4*) [source]

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*) [source]

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')
```

terminal(*module_color='default'*, *background='reverse'*, *quiet_zone=4*) [source]

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)
```

text(*quiet_zone=4*) [source]

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 *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)
```

xbm(*scale=1, quiet_zone=4*) [source]

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 pyqrcode
>>> import tkinter
>>> code = pyqrcode.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 `get_png_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.