Pydenticon Documentation

Release 0.3.1

Branko Majic

Sep 04, 2017

Contents

Pydenticon is a small utility library that can be used for deterministically generating identicons based on the hash of provided data.

The implementation is a port of the Sigil identicon implementation from:

- https://github.com/cupcake/sigil/

CHAPTER 1

Support

In case of problems with the library, please do not hestitate to contact the author at **pydenticon (at) majic.rs**. The library itself is hosted on Github, and on author's own websites:

- https://github.com/azaghal/pydenticon
- https://code.majic.rs/pydenticon
- https://projects.majic.rs/pydenticon

Contents:

About Pydenticon

Pydenticon is a small utility library that can be used for deterministically generating identicons based on the hash of provided data.

The implementation is a port of the Sigil identicon implementation from:

- https://github.com/cupcake/sigil/

Why was this library created?

A number of web-based applications written in Python have a need for visually differentiating between users by using avatars for each one of them.

This functionality is particularly popular with comment-posting since it increases the readability of threads.

The problem is that lots of those applications need to allow anonymous users to post their comments as well. Since anonymous users cannot set the avatar for themselves, usually a random avatar is created for them instead.

There is a number of free (as in free beer) services out there that allow web application developers to create such avatars. Unfortunately, this usually means that the users visiting websites based on those applications are leaking information about their browsing habits etc to these third-party providers.

Pydenticon was written in order to resolve such an issue for one of the application (Django Blog Zinnia, in particular), and to allow the author to set up his own avatar/identicon service.

Features

Pydenticon has the following features:

- Compatible with Sigil implementation (https://github.com/cupcake/sigil/) if set-up with right parameters.
- Creates vertically symmetrical identicons of any rectangular shape and size.
- Uses digests of passed data for generating the identicons. * Automatically detects if passed data is hashed already or not. * Custom digest implementations can be passed to identicon generator (defaults to 'MD5').
- Support for multiple image formats. * PNG * ASCII
- Foreground colour picked from user-provided list.
- Background colour set by the user.
- Ability to invert foreground and background colour in the generated identicon.
- Customisable padding around generated identicon using the background colour (foreground if inverted identicon was requested).

Installation

Pydenticon can be installed through one of the following methods:

- Using *pip*, which is the easiest and recommended way for production websites.
- Manually, by copying the necessary files and installing the dependencies.

Requirements

The main external requirement for Pydenticon is Pillow, which is used for generating the images.

Using pip

In order to install latest stable release of Pydenticon using *pip*, run the following command:

```
pip install pydenticon
```

In order to install the latest development version of Pydenticon from Github, use the following command:

```
pip install -e git+https://github.com/azaghal/pydenticon#egg=pydenticon
```

Manual installation

If you wish to install Pydenticon manually, make sure that its dependencies have been met first, and then simply copy the `pydenticon` directory (that contains the `__init__.py` file) somewhere on the Python path.

Usage

Pydenticon provides simple and straightforward interface for setting-up the identicon generator, and for generating the identicons.

Instantiating a generator

The starting point is to create a generator instance. Generator implements interface that can be used for generating the identicons.

In its simplest form, the generator instances needs to be passed only the size of identicon in blocks (first parameter is width, second is height):

```
# Import the library.
import pydenticon

# Instantiate a generator that will create 5x5 block identicons.
generator = pydenticon.Generator(5, 5)
```

The above example will instantiate a generator that can be used for producing identicons which are 5x5 blocks in size, using the default values for digest (*MD5*), foreground colour (*black*), and background colour (*white*).

Alternatively, you may choose to pass in a different digest algorithm, and foreground and background colours:

```
# Import the libraries.
import pydenticon
import hashlib

# Set-up a list of foreground colours (taken from Sigil).
foreground = [ "rgb(45,79,255)",
               "rgb(254,180,44)",
               "rgb(226,121,234)",
               "rgb(30,179,253)",
               "rgb(232,77,65)",
               "rgb(49,203,115)",
               "rgb(141,69,170)" ]

# Set-up a background colour (taken from Sigil).
background = "rgb(224,224,224)"

# Instantiate a generator that will create 5x5 block identicons using SHA1
# digest.
generator = pydenticon.Generator(5, 5, digest=hashlib.sha1,
                                 foreground=foreground, background=background)
```

Generating identicons

With generator initialised, it's now possible to use it to create the identicons.

The most basic example would be creating an identicon using default padding (no padding) and output format ("png"), without inverting the colours (which is also the default):

```
# Generate a 240x240 PNG identicon.
identicon = generator.generate("john.doe@example.com", 240, 240)
```

The result of the `generate()` method will be a raw representation of an identicon image in requested format that can be written-out to file, sent back as an HTTP response etc.

Usually it can be nice to have some padding around the generated identicon in order to make it stand-out better, or maybe to invert the colours. This can be done with:

```
# Set-up the padding (top, bottom, left, right) in pixels.
padding = (20, 20, 20, 20)

# Generate a 200x200 identicon with padding around it, and invert the
# background/foreground colours.
identicon = generator.generate("john.doe@example.com", 200, 200,
                               padding=padding, inverted=True)
```

Finally, the resulting identicons can be in different formats:

```
# Create identicon in PNG format.
identicon_png = generator.generate("john.doe@example.com", 200, 200,
                                   output_format="png")
# Create identicon in ASCII format.
identicon_ascii = generator.generate("john.doe@example.com", 200, 200,
                                     output_format="ascii")
```

Supported output formats are dependant on the local Pillow installation. For exact list of available formats, have a look at Pillow documentation. The `ascii` format is the only format explicitly handled by the *Pydenticon* library itself (mainly useful for debugging purposes).

Using the generated identicons

Of course, just generating the identicons is not that fun. They usually need either to be stored somewhere on disk, or maybe streamed back to the user via HTTP response. Since the generate function returns raw data, this is quite easy to achieve:

```
# Generate same identicon in three different formats.
identicon_png = generator.generate("john.doe@example.com", 200, 200,
                                   output_format="png")
identicon_gif = generator.generate("john.doe@example.com", 200, 200,
                                   output_format="gif")
identicon_ascii = generator.generate("john.doe@example.com", 200, 200,
                                     output_format="ascii")

# Identicon can be easily saved to a file.
f = open("sample.png", "wb")
f.write(identicon_png)
f.close()

f = open("sample.gif", "wb")
f.write(identicon_gif)
f.close()

# ASCII identicon can be printed-out to console directly.
print identicon_ascii
```

Working with transparency

Note: New in version `0.3`.

Warning: The only output format that properly supports transparency at the moment is `PNG`. If you are using anything else, transparency will not work.

If you ever find yourself in need of having a transparent background or foreground, you can easily do this using the syntax `rgba(224,224,224,0)`. All this does is effectively adding alpha channel to selected colour.

The alpha channel value ranges from `0` to `255`, letting you specify how much transparency/opaqueness you want. For example, to have it at roughly 50% (more like at `50.2%` since you can't use fractions), you would simply specify value as `rgba(224,224,224,128)`.

Full example

Finally, here is a full example that will create a number of identicons and output them in PNG format to local directory:

```python
#!/usr/bin/env python

# Import the libraries.
import pydenticon
import hashlib

# Set-up some test data.
users = ["alice", "bob", "eve", "dave"]

# Set-up a list of foreground colours (taken from Sigil).
foreground = [ "rgb(45,79,255)",
               "rgb(254,180,44)",
               "rgb(226,121,234)",
               "rgb(30,179,253)",
               "rgb(232,77,65)",
               "rgb(49,203,115)",
               "rgb(141,69,170)" ]

# Set-up a background colour (taken from Sigil).
background = "rgb(224,224,224)"

# Set-up the padding (top, bottom, left, right) in pixels.
padding = (20, 20, 20, 20)

# Instantiate a generator that will create 5x5 block identicons using SHA1
# digest.
generator = pydenticon.Generator(5, 5, foreground=foreground,
                                 background=background)

for user in users:
  identicon = generator.generate(user, 200, 200, padding=padding,
                                 output_format="png")

  filename = user + ".png"
  with open(filename, "wb") as f:
      f.write(identicon)
```

Algorithm

A generated identicon can be described as one big rectangle divided into `rows x columns` rectangle blocks of equal size, where each block can be filled with the foreground colour or the background colour. Additionally, the whole identicon is symmetrical to the central vertical axis, making it much more aesthetically pleasing.

The algorithm used for generating the identicon is fairly simple. The input arguments that determine what the identicon will look like are:

- Size of identicon in blocks (`rows x columns`).
- Algorithm used to create digests out of user-provided data.
- List of colours used for foreground fill (foreground colours). This list will be referred to as `foreground_list`.
- Single colour used for background fill (background colour). This colour wil be referred to as `background`.
- Whether the foreground and background colours should be inverted (swapped) or not.
- Data passed to be used for digest.

The first step is to generate a *digest* out of the passed data using the selected digest algorithm. This digest is then split into two parts:

- The first byte of digest (`f`, for foreground) is used for determining the foreground colour.
- The remaining portion of digest (`l`, for layout) is used for determining which blocks of identicon will be filled using foreground and background colours.

In order to select a `foreground` colour, the algorithm will try to determine the index of the colour in the `foreground_list` by doing modulo division of the first byte's integer value with number of colours in `foreground_list`:

```
foreground = foreground_list[int(f) % len(foreground_list)]
```

The layout of blocks (which block gets filled with foreground colour, and which block gets filled with background colour) is determined by the bit values of remaining portion of digest (`l`). This remaining portion of digest can also be seen as a list of bits. The bit positions would range from `0` to `b` (where the size of `b` would depend on the digest algoirthm that was picked).

Since the identicon needs to be symmetrical, the number of blocks for which the fill colour needs to be calculated is equal to `rows * (columns / 2 + columns % 2)`. I.e. the block matrix is split in half vertically (if number of columns is odd, the middle column is included as well).

Those blocks can then be marked with whole numbers from `0` to `c` (where `c` would be equal to the above formula - `rows * (columns / 2 + columns % 2)`). Number `0` would correspond to first block of the first half-row, `1` to the first block of the second row, `2` to the first block of the third row, and so on to the first block of the last half-row. Then the blocks in the next column would be indexed with numbers in a similar (incremental) way.

With these two numbering methods in place (for digest bits and blocks of half-matrix), every block is assigned a bit that has the same position number.

If no inversion of foreground and background colours was requested, bit value of `1` for a cell would mean the block should be filled with foreground colour, while value of `0` would mean the block should be filled with background colour.

If an inverted identicon was requested, then `1` would correspond to background colour fill, and `0` would correspond to foreground colour fill.

Examples

An identicon should be created with the following parameters:

- Size of identicon in blocks is `5 x 5` (a square).
- Digest algorithm is *MD5*.
- Five colours are used for identicon foreground (`0` through `4`).
- Some background colour is selected (marked as `b`).
- Foreground and background colours are not to be inverted (swapped).
- Data used for digest is `branko`.

MD5 digest for data (`branko`) would be (reperesented as hex value) equal to `d41c0e80c44173dcf7575745bdddb704`.

In other words, 16 bytes would be present with the following hex values:

```
d4 1c 0e 80 c4 41 73 dc f7 57 57 45 bd dd b7 04
```

Following the algorithm, the first byte (`d4`) is used to determine which foreground colour to use. `d4` is equal to `212` in decimal format. Divided by modulo `5` (number of foreground colours), the resulting index of foreground colour is `2` (third colour in the foreground list).

The remaining 15 bytes will be used for figuring out the layout. The representation of those bytes in binary format would look like this (5 bytes per row):

```
00011100 00001110 10000000 11000100 01000001
01110011 11011100 11110111 01010111 01010111
01000101 10111101 11011101 10110111 00000100
```

Since identicon consits out of 5 columns and 5 rows, the number of bits that's needed from `1` for the layout would be `5 * (5 / 2 + 5 % 2) == 15`. This means that the following bits will determine the layout of identicon (whole first byte, and 7 bits of the second byte):

```
00011100 0000111
```

The half-matrix would therefore end-up looking like this (5 bits per column for 5 blocks per column):

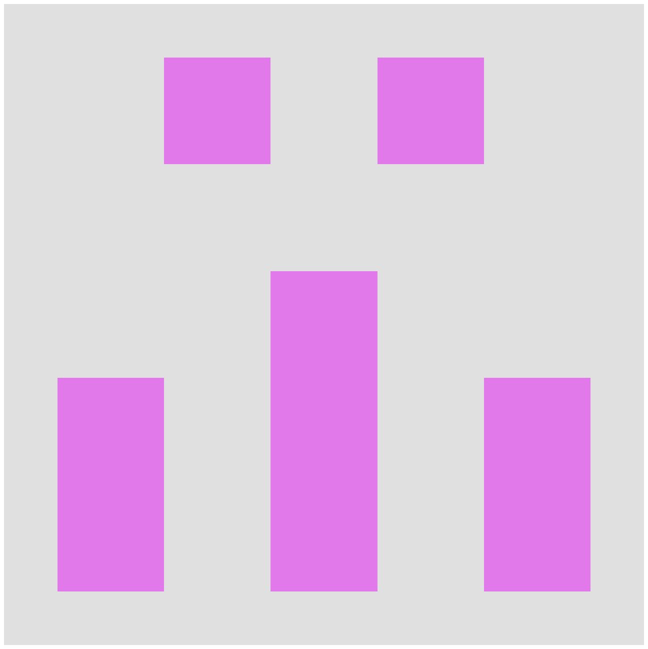
```
010
000
001
101
101
```

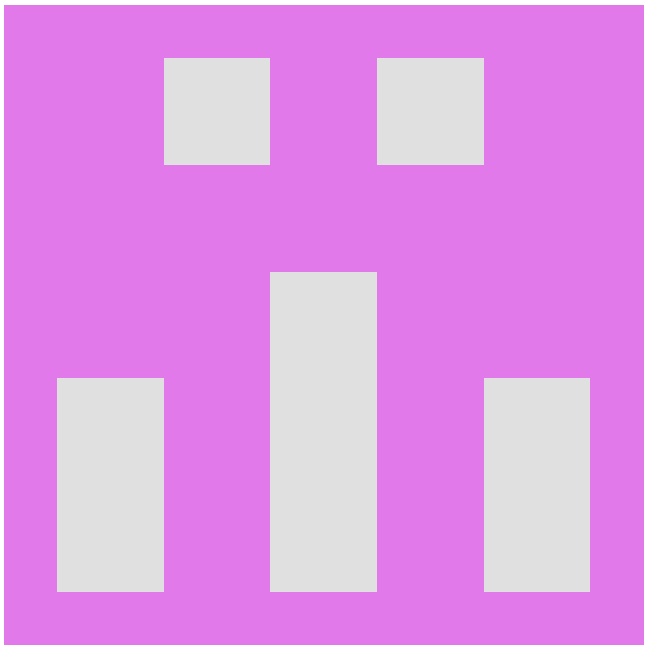
The requested identicon is supposed to have 5 block columns, so a reflection will be applied to the first and second column, with third column as center of the symmetry. This would result in the following ideticon matrix:

```
01010
00000
00100
10101
10101
```

Since no inversion was requested, `1` would correspond to calculated foreground colour, while `0` would correspond to provided background colour.

To spicen the example up a bit, here is what the above identicon would look like in regular and inverted variant (with some sample foreground colours and a bit of padding):





Limitations

There's some practical limitations to the algorithm described above.

The first limitation is the maximum number of different foreground colours that can be used for identicon generation. Since a single byte (which is used to determining the colour) can represent 256 values (between 0 and 255), there can be no more than 256 colours passed to be used for foreground of the identicon. Any extra colours passed above that count would simply be ignored.

The second limitation is that the maximum dimensions (in blocks) of a generated identicon depend on digest algorithm used. In order for a digest algorithm to be able to satisfy requirements of producing an identcion with `rows` number of rows, and `columns` number of columns (in blocks), it must be able to produce at least the following number of bits (i.e. the number of bits equal to the number of blocks in the half-matrix):

```
rows * (columns / 2 + columns % 2) + 8
```

The expression is the result of vertical symmetry of identicon. Only the columns up to, and including, the middle one middle one (`(columns / 2 + colums % 2)`) need to be processed, with every one of those columns having `row` rows (`rows *`). Finally, an extra 8 bits (1 byte) are necessary for determining the foreground colour.

Privacy

It is fundamentally important to understand the privacy issues if using Pydenticon in order to generate uniquelly identifiable avatars for users leaving the comments etc.

The most common way to expose the identicons is by having a web application generate them on the fly from data that is being passed to it through HTTP GET requests. Those GET requests would commonly include either the raw data, or data as hex string that is then used to generate an identicon. The URLs for GET requests would most commonly be made as part of image tags in an HTML page.

The data passed needs to be unique in order to generate distinct identicons. In most cases the data used will be either name or e-mail address that the visitor posting the comment fills-in in some field. That being said, e-mails usually provide a much better identifier than name (especially if the website verifies the comments through by sending-out e-mails).

Needless to say, in such cases, especially if the website where the comments are being posted is public, using raw data can completely reveal the identity of the user. If e-mails are used for generating the identicons, the situation is even worse, since now those e-mails can be easily harvested for spam purposes. Using the e-mails also provides data mining companies with much more reliable user identifier that can be coupled with information from other websites.

Therefore, it is highly recommended to pass the data to web application that generates the identicons using **hex digest only**. I.e. **never** pass the raw data.

Although passing hash instead of real data as part of the GET request is a good step forward, it can still cause problems since the hashses can be collected, and then used in conjunction with rainbow tables to identify the original data. This is particularly problematic when using hex digests of e-mail addresses as data for generating the identicon.

There's two feasible approaches to resolve this:

- Always apply *salt* to user-identifiable data before calculating a hex digest. This can hugely reduce the efficiency of brute force attacks based on rainbow tables (althgouh it will not mitigate it completely).
- Instead of hashing the user-identifiable data itself, every time you need to do so, create some random data instead, hash that random data, and store it for future use (cache it), linking it to the original data that it was generated for. This way the hex digest being put as part of an image link into HTML pages is not derived in any way from the original data, and can therefore not be used to reveal what the original data was.

Keep in mind that using identicons will inevitably still allow people to track someone's posts across your website. Identicons will effectively automatically create pseudonyms for people posting on your website. If that may pose a problem, it might be better not to use identicons at all.

Finally, small summary of the points explained above:

- Always use hex digests in order to retrieve an identicon from a server.
- Instead of using privately identifiable data for generating the hex digest, use randmoly generated data, and associate it with privately identifiable data. This way hex digest cannot be traced back to the original data through brute force or rainbow tables.

- If unwilling to generate and store random data, at least make sure to use salt when hashing privately identifiable data.

API Reference

class `pydenticon.`**`Generator`**(*rows, columns, digest=<built-in function openssl_md5>, foreground=['#000000'], background='#ffffff'*)

Factory class that can be used for generating the identicons deterministically based on hash of the passed data.

Resulting identicons are images of requested size with optional padding. The identicon (without padding) consists out of M x N blocks, laid out in a rectangle, where M is the number of blocks in each column, while N is number of blocks in each row.

Each block is a smallself rectangle on its own, filled using the foreground or background colour.

The foreground is picked randomly, based on the passed data, from the list of foreground colours set during initialisation of the generator.

The blocks are always laid-out in such a way that the identicon will be symterical by the Y axis. The center of symetry will be the central column of blocks.

Simply put, the generated identicons are small symmetric mosaics with optional padding.

`generate`(*data, width, height, padding=(0, 0, 0, 0), output_format='png', inverted=False*)

Generates an identicon image with requested width, height, padding, and output format, optionally inverting the colours in the indeticon (swapping background and foreground colours) if requested.

Arguments:

data - Hashed or raw data that will be used for generating the identicon.

width - Width of resulting identicon image in pixels.

height - Height of resulting identicon image in pixels.

padding - Tuple describing padding around the generated identicon. The tuple should consist out of four values, where each value is the number of pixels to use for padding. The order in tuple is: top, bottom, left, right.

output_format - Output format of resulting identicon image. Supported formats are anything that is supported by Pillow, plus a special "ascii" mode.

inverted - Specifies whether the block colours should be inverted or not. Default is False.

Returns:

Byte representation of an identicon image.

Testing

Pydenticon includes a number of unit tests which are used for regression testing. The tests are fairly comprehensive, and also include comparison of Pydenticon-generated identicons against a couple of samples generated by Sigil.

Tests depend on the following additional libraries:

- Mock

Test dependencies will be automatically downloaded when running the tests if they're not present.

Pydenticon tests can be run with the following command:

```
python setup.py test
```

Release Notes

0.3.1

Minor bug-fixes.

Bug fixes:

- PYD-8 - Cannot generate identicons in JPEG format when using Pillow >= 4.2.0

0.3

Update introducing support for more output formats and ability to use transparency for PNG identicons.

New features:

- PYD-6: Add support for having transparent backgrounds in identicons

  Ability to use alpha-channel specification in PNG identicons to obtain complete or partial transparency. Works for both background and foreground colour.

- PYD-7: Ability to specify image format

  Ability to specify any output format supported by the Pillow library.

0.2

A small release that adds support for Python 3 in addition to Python 2.7.

New features:

- PYD-5: Add support for Python 3.x

  Support for Python 3.x, in addition to Python 2.7.

0.1.1

This is a very small release feature-wise, with a single bug-fix.

New features:

- PYD-3: Initial tests

  Unit tests covering most of the library functionality.

Bug fixes:

- PYD-4: Identicon generation using pre-hashed data raises ValueError

  Fixed some flawed logic which prevented identicons to be generated from existing hashes.

0.1

Initial release of Pydenticon. Implemented features:

- Supported parameters for identicon generator (shared between multiple identicons): * Number of blocks in identicon (rows and columns). * Digest algorithm. * List of foreground colours to choose from. * Background colour.
- Supported parameters when generating induvidual identicons: * Data that should be used for identicon generation. * Width and height of resulting image in pixels. * Padding around identicon (top, bottom, left, right). * Output format. * Inverted identicon (swaps foreground with background).
- Support for PNG and ASCII format of resulting identicons.
- Full documentation covering installation, usage, algorithm, privacy. API reference included as well.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index