# libimagequant Python Bindings

***Release 2.17.0.0***

**Feb 04, 2022**

# Contents:

Welcome to the documentation for the unofficial Python bindings for libimagequant.

These bindings are designed to be Pythonic, yet still faithful to the C API. Almost every C function can be used through the bindings. The Python classes correspond directly to C structs, and each Python function represents one C function. However, some changes have been made:

- All functions have been made into class methods.
- Functions that are effectively getters and setters for struct members are represented as class properties.
- Values that are semantically boolean but are of the `int` type in C are given the Python `bool` type.
- Error-code return values are instead expressed by raising exceptions (see *Exceptions*).
- A few functions – mostly ones that don't make much sense in Python – are not supported (see *Functions with no direct Python equivalent*).

This documentation is intentionally terse, so as to avoid duplicating the information in the official C API documentation. The recommended way to use this page is to first peruse the official libimagequant C API documentation to see how you could accomplish your goals in C, and to then search for the C function names here to find the equivalent Python APIs.

You may want to take a look at *Examples*, *Installation*, or the *API reference*.

**Note:** You might also be interested in the companion library libimagequant_integrations, which provides helper functions for using libimagequant with many other Python libraries used for imagery.

# CHAPTER 1

# Indices and tables

- genindex
- modindex
- search

CHAPTER 2

# Examples

*Note:* instead of copypasting these examples in order to use libimagequant with PyPNG or other imagery modules, consider using the libimagequant_integrations library, which provides robust conversion functions for you.

Here's the simplest useful example, which uses PyPNG for loading/saving PNGs:

```python
import libimagequant as liq
import png

# Load the image with PyPNG
img = png.Reader(filename='input.png')
width, height, data, info = img.read_flat()

# Create libimagequant Attr and Image objects from it
attr = liq.Attr()
input_image = attr.create_rgba(data, width, height, info.get('gamma', 0))

# Quantize
result = input_image.quantize(attr)

# Get the quantization result
out_pixels = result.remap_image(input_image)
out_palette = result.get_palette()

# Save it
writer = png.Writer(input_image.width, input_image.height, palette=out_palette)
with open('output.png', 'wb') as f:
    writer.write_array(f, out_pixels)
```

And here's a port of example.c from the libimagequant repository:

```python
import sys

import libimagequant as liq
import png
```

(continues on next page)

(continued from previous page)

```
def main(argv):
    if len(argv) < 2:
        print('Please specify a path to a PNG file', file=sys.stderr)
        return 1

    input_png_file_path = argv[1]

    # Load PNG file and decode it as raw RGBA pixels
    # This uses the PyPNG library for PNG reading (not part of libimagequant)

    reader = png.Reader(filename=input_png_file_path)
    width, height, input_rgba_pixels, info = reader.read_flat()

    # Use libimagequant to make a palette for the RGBA pixels

    attr = liq.Attr()
    input_image = attr.create_rgba(input_rgba_pixels, width, height, info.get('gamma',
↪ 0))

    result = input_image.quantize(attr)

    # Use libimagequant to make new image pixels from the palette

    result.dithering_level = 1.0

    raw_8bit_pixels = result.remap_image(input_image)
    palette = result.get_palette()

    # Save converted pixels as a PNG file
    # This uses the PyPNG library for PNG writing (not part of libimagequant)
    writer = png.Writer(input_image.width, input_image.height, palette=palette)

    output_png_file_path = 'quantized_example.png'
    with open(output_png_file_path, 'wb') as f:
        writer.write_array(f, raw_8bit_pixels)

    print('Written ' + output_png_file_path)

    # Done.

main(sys.argv)
```

# CHAPTER 3

# Installation

Builds (fully unit-tested) are provided for supported versions of CPython 3 (3.6 through 3.10, at the time of this writing) on the following platforms:

- x86_64 Windows
- x86 (32-bit) Windows
- x86_64 macOS
- x86_64 Linux (for both the "manylinux2014" platform and a PEP-600-compatible manylinux platform)

A source distribution ("sdist") is also available, which should be compatible with PyPy, as well as other platforms and architectures.

The recommended way to install is through pip. You can try running:

```
pip install libimagequant
```

If that doesn't work, you might have better luck with either of:

```
python3 -m pip install libimagequant

py -3 -m pip install libimagequant
```

If for some reason you'd instead like to install from source manually (such as for debugging), read on.

## 3.1 Building from source

To build from source manually, begin by cloning or downloading the repository.

If desired, you can replace the `libimagequant` folder with the latest libimagequant source code from its own repository.

Install `cffi`, `setuptools` and `wheel` on the Python interpreter you want the bindings to be built against. For example,

```
python3 -m pip install --upgrade cffi setuptools wheel
```

Navigate (in a terminal) to the `bindings` directory, and run `setup.py bdist_wheel` with the Python interpreter you want the bindings to be built against. For example,

```
python3 setup.py bdist_wheel
```

This should create (among other things) a `dist` folder with a `.whl` (wheel) file inside. You can now install that wheel file with pip, or distribute it.

CHAPTER 4

# API reference

## 4.1 Exceptions

Many functions in libimagequant's C API use `liq_error` enum return values to indicate success or errors. Since it is more Pythonic to use exceptions for this, the Python bindings for those functions convert those return values to exceptions, which you can catch using `try`/`except`. The following table outlines how they're mapped:

| `liq_error` value | Python exception |
|---|---|
| `LIQ_OK` | *(n/a)* |
| `LIQ_QUALITY_TOO_LOW` | `libimagequant.QualityTooLowError` |
| `LIQ_VALUE_OUT_OF_RANGE` | `ValueError` |
| `LIQ_OUT_OF_MEMORY` | `MemoryError` |
| `LIQ_ABORTED` | `libimagequant.AbortedError` |
| `LIQ_BITMAP_NOT_AVAILABLE` | `libimagequant.BitmapNotAvailableError` |
| `LIQ_BUFFER_TOO_SMALL` | `libimagequant.BufferTooSmallError` |
| `LIQ_INVALID_POINTER` | `RuntimeError` |
| `LIQ_UNSUPPORTED` | `libimagequant.UnsupportedError` |

## 4.2 Constants

**`libimagequant.LIQ_VERSION and libimagequant.LIQ_VERSION_STRING`**

Information about the version of **libimagequant** currently in use.

Depending on your use case, you may want to use `BINDINGS_VERSION` and `BINDINGS_VERSION_STRING` instead.

Python equivalents of `LIQ_VERSION` and `LIQ_VERSION_STRING`.

**`libimagequant.BINDINGS_VERSION and libimagequant.BINDINGS_VERSION_STRING`**

Information about the version of the **Python bindings** currently in use.

The bindings version is the version of libimagequant the bindings were designed for, with an additional version segment (usually `.0`). For example, for the bindings release designed for libimagequant 2.12.5, `BINDINGS_VERSION` and `BINDINGS_VERSION_STRING` would be 2120500 and `'2.12.5.0'`, respectively.

This will often match `LIQ_VERSION` and `LIQ_VERSION_STRING` (up to the extra segment), but is not guaranteed to always do so.

Depending on your use case, you may want to use `LIQ_VERSION` and `LIQ_VERSION_STRING` instead.

## 4.3 Classes

**class** `libimagequant.`**`Attr`**

Python equivalent of the `liq_attr` struct.

The constructor for this class is the equivalent of `liq_attr_create()`. `liq_attr_destroy()` is handled automatically.

**`max_colors`**

Python equivalent of `liq_get_max_colors()` and `liq_set_max_colors()`.

**Type** `int`

**`speed`**

Python equivalent of `liq_get_speed()` and `liq_set_speed()`.

**Type** `int`

**`min_opacity`**

Python equivalent of `liq_get_min_opacity()` and `liq_set_min_opacity()`.

**Type** `int`

**`min_posterization`**

Python equivalent of `liq_get_min_posterization()` and `liq_set_min_posterization()`.

**Type** `int`

**`min_quality`**

Python equivalent of `liq_get_min_quality()` and (along with *`max_quality`*) `liq_set_quality()`.

**Type** `int`

**`max_quality`**

Python equivalent of `liq_get_max_quality()` and (along with *`min_quality`*) `liq_set_quality()`.

**Type** `int`

**`last_index_transparent`**

Python equivalent of `liq_set_last_index_transparent()`.

For consistency with the C API, this is a write-only property.

**Note:** Since the only meaningful values for this variable in the C API are “zero” and “non-zero,” it is presented as a `bool` in these Python bindings.

**Type** `bool`

**`copy`**`()` → Attr
Python equivalent of `liq_attr_copy()`.

**Returns** A copy of this object.

**Return type** *`libimagequant.Attr`*

**`create_rgba`**`(`*bitmap: bytes*, *width: int*, *height: int*, *gamma: float*`)` → Image
Python equivalent of `liq_image_create_rgba()`.

**Returns** The new image created from the provided data.

**Return type** *`libimagequant.Image`*

**`set_log_callback`**`(`*log_callback_function: Callable[[Attr, str, object], None], user_info: object*`)`
Python equivalent of `liq_set_log_callback()`.

The signature of the callback function should be `callback(attr: Attr, message: str, user_info: object)`.

The `user_info` parameter can be any Python object, which will be passed to the callback as its third argument.

Call this function with `log_callback_function = None` to clear the callback.

**`set_progress_callback`**`(`*progress_callback_function: Callable[[float, object], bool], user_info: object*`)`
Python equivalent of `liq_attr_set_progress_callback()`.

The signature of the callback function should be `callback(progress_percent: float, user_info: object) -> bool`. If it returns `False`, the quantization operation will be aborted (causing `AbortedException` to be raised); thus, you should normally return `True` from the callback in order for the operation to proceed.

The `user_info` parameter can be any Python object, which will be passed to the callback as its third argument.

Call this function with `progress_callback_function = None` to clear the callback.

**class** `libimagequant.`**`Histogram`**`(`*attr: Attr*`)`
Python equivalent of the `liq_histogram` struct.

The constructor for this class is the equivalent of `liq_histogram_create()`. `liq_histogram_destroy()` is handled automatically.

**`add_image`**`(`*attr: Attr*, *image: Image*`)`
Python equivalent of `liq_histogram_add_image()`.

**`add_colors`**`(`*attr: Attr, entries: List[HistogramEntry], gamma: float*`)`
Python equivalent of `liq_histogram_add_colors()`.

**`add_fixed_color`**`(`*color: Color*, *gamma: float*`)`
Python equivalent of `liq_histogram_add_fixed_color()`.

**`quantize`**`(`*options: Attr*`)` → Result
Python equivalent of `liq_histogram_quantize()`.

**Returns** The result of the quantization.

**Return type** *`libimagequant.Result`*

**class** `libimagequant.`**`HistogramEntry`**`(`*color: Color*, *count: int*`)`
Python equivalent of the `liq_histogram_entry` struct.

**`color`**
Python equivalent of the `liq_histogram.color` member.

**Type** *libimagequant.Color*

**count**
Python equivalent of the `liq_histogram.count` member.

**Type** `int`

**class** libimagequant.**Image**
Python equivalent of the `liq_image` struct.

This class cannot be instantiated directly. Use `Image.create_rgba()` to create it.

`liq_image_destroy()` is handled automatically.

**width**
Python equivalent of `liq_image_get_width()`.

This is a read-only property.

**Type** `int`

**height**
Python equivalent of `liq_image_get_height()`.

This is a read-only property.

**Type** `int`

**background**
Python equivalent of `liq_image_set_background()`.

For consistency with the C API, this is a write-only property.

**Type** *libimagequant.Image*

**importance_map**
Python equivalent of `liq_image_set_importance_map()`.

For consistency with the C API, this is a write-only property.

**Type** `bytes`

**add_fixed_color**(*color: Color*)
Python equivalent of `liq_image_add_fixed_color()`.

**quantize**(*options: Attr*) → Result
Python equivalent of `liq_image_quantize()`.

**Returns** The result of the quantization.

**Return type** *libimagequant.Result*

**class** libimagequant.**Result**
Python equivalent of the `liq_result` struct.

This class cannot be instantiated directly. Use `Histogram.quantize()` or `Image.quantize()` to create it.

`liq_result_destroy()` is handled automatically.

**dithering_level**
Python equivalent of `liq_set_dithering_level()`.

For consistency with the C API, this is a write-only property.

**Type** `float`

**`output_gamma`**
Python equivalent of `liq_get_output_gamma()` and `liq_set_output_gamma()`.

**Type** `float`

**`quantization_error`**
Python equivalent of `liq_get_quantization_error()`.

This is a read-only property.

**Type** `float`

**`quantization_quality`**
Python equivalent of `liq_get_quantization_quality()`.

This is a read-only property.

**Type** `int`

**`remapping_error`**
Python equivalent of `liq_get_remapping_error()`.

This is a read-only property.

**Type** `float`

**`remapping_quality`**
Python equivalent of `liq_get_remapping_quality()`.

This is a read-only property.

**Type** `int`

**`get_palette`**`()` → List[Color]
Python equivalent of `liq_get_palette()`.

**Returns** The list of colors.

**Return type** `list` of *`libimagequant.Color`*s

**`remap_image`**(*input_image: Image*) → bytes
Python equivalent of `liq_write_remapped_image()`.

**Returns** The pixel data for the remapped image.

**Return type** `bytes`

**`set_progress_callback`**(*progress_callback_function: Callable[[float, object], bool], user_info: object*)
Python equivalent of `liq_result_set_progress_callback()`.

The signature of the callback function should be `callback(progress_percent: float, user_info: object) -> bool`. If it returns `False`, the remapping operation will be aborted (causing `AbortedException` to be raised); thus, you should normally return `True` from the callback in order for the operation to proceed.

The `user_info` parameter can be any Python object, which will be passed to the callback as its third argument.

Call this function with `progress_callback_function = None` to clear the callback.

**`class`** `libimagequant.`**`Color`**
Python equivalent of the `liq_color` struct.

This is simply a collections.namedtuple with `r`, `g`, `b`, and `a` fields.

Please note that the equivalent of a `liq_palette` struct in these bindings is a `list` of instances of this class.

CHAPTER 5

# Functions with no direct Python equivalent

- `liq_attr_create_with_allocator()`

  Although "custom allocators" aren't *completely* meaningless in Python (in the context of cffi, in particular), it's an extremely uncommon case.

  If you have a legitimate need for this feature, please open an issue (or, better, a pull request!). For 99% of cases, `Attr`'s default constructor (corresponding to `liq_attr_create()`) should suffice.

- `liq_set_log_flush_callback()`

  This is unsupported due to issues that arise due to Python's garbage collection. Since functions in Python are objects that get garbage-collected like all other types, there is no guarantee that the callback will actually still exist when the `Attr` object is deleted. This can lead to very weird and inconsistent issues.

  Since libimagequant is totally synchronous, the recommended workaround is to simply flush any logging resources after you finish using your libimagequant objects.

- `liq_image_create_rgba_rows()` and `liq_image_create_custom()`

  These are unsupported because Python does not allow for the fine-grained raw pointer access that would make these functions useful.

  Use `Image.create_rgba()` (corresponding to `liq_image_create_rgba()`) instead.

- `liq_image_set_memory_ownership()`

  This is unsupported because it's too low-level of a concern to expose to Python programs. Ensuring that memory is managed properly is the responsibility of the bindings themselves, not your application.

- `liq_write_remapped_image_rows()`

  This is unsupported because Python does not allow for the fine-grained raw pointer access that would make it useful.

  Use `Result.remap_image()` (corresponding to `liq_write_remapped_image()`) instead.

- `liq_version()`

  Use `LIQ_VERSION` or `BINDINGS_VERSION` instead, depending on if you need to check the libimagequant version or the Python bindings version.

- `liq_quantize_image()`

  This is unsupported because it is deprecated in the C API. Use `Image.quantize()` (corresponding to `liq_image_quantize()`) instead.

# Index