Willow Documentation

Release 1.4

Torchbox

May 26, 2020

Contents

1 Index **3**
- 1.1 Installation . . . 3
- 1.2 Concepts . . . 3
- 1.3 Usage guide . . . 4
- 1.4 Reference . . . 11
- 1.5 Changelog . . . 15

Index **19**

Willow is a pure Python library that aims to unite many Python imaging libraries under a single interface.

Out of the box, Willow can work with Pillow, Wand or OpenCV. None of these image libraries are required (but you should have either Pillow or Wand installed to use most features). It also has a plugin interface which allows you to add support for more libraries, image formats and operations.

CHAPTER 1

Index

1.1 Installation

Willow supports Python 2.7+ and 3.4+. It's a pure-python library with no hard dependencies so doesn't require a C compiler for a basic installation.

1.1.1 Installation using `pip`

```
pip install Willow
```

1.1.2 Installing underlying libraries

In order for most features of Willow to work, you need to install either Pillow or Wand.

- Pillow installation
- Wand installation

Note that Pillow doesn't support animated GIFs and Wand isn't as fast. Installing both will give best results.

1.2 Concepts

1.2.1 Image classes

An image can either be a file, an image loaded into an underlying library or a simple buffer of pixels. Each of these states has its own Python class (subclass of `willow.image.Image`).

For example `JPEGImageFile`, `PillowImage` and `RGBAImageBuffer` are three of the image classes in Willow.

1.2.2 Operations

These are functions that perform actions on an image in a particular state. For example, `resize` and `crop`.

Operations can either be defined as methods on the image class or as functions registered separately.

All operations are registered in a central registry and will appear as a method on all other image classes. If it's called from a class that doesn't implement the operation, the image will be automatically converted to the nearest image class that supports it and the operation is run on that.

Operations that alter an image return a new image object instead of altering the source one. This also means that if a conversion took place, the new image's class would be different.

1.2.3 Converters

These are functions that convert an image between two image classes. For example, a converter from `JPEGImageFile` to `PillowImage` would simply be a function that calls `PIL.Image.open` on the underlying file to get a Pillow image.

Like operations, these can either be methods on the image class or registered separately.

Each converter has a cost which helps Willow decide which is the best available image library to use for a particular file format.

1.2.4 Registry

The registry is where all image classes, operations and converters are registered. It contains methods to allow you to register new items and even override existing ones.

It also is responsible for finding operations and planning routes between image classes.

1.2.5 Plugins

These are used to group related image classes, operations and converters together allowing them to be registered as a single unit.

The convention within Willow is to create a single plugin for each underlying library. The default ones are "pillow", "wand" and "opencv".

Plugins can be registered even if the underlying library is not installed. This allows Willow to generate a useful error message if an operation is requested that only exists in a plugin without an underlying library.

1.3 Usage guide

If you're looking at Willow for the first time, have a look at the *concepts* and *installation guide* first.

This section describes general, everyday usage of the Willow library.

1.3.1 Index

Opening images

Images can be either opened from a file or an existing Pillow/Wand object.

From a file

To open an image, call `Image.open()` passing in a file-like object that contains the image data.

```
from willow.image import Image

with open('test.jpg', 'rb') as f:
    i = Image.open(f)

    isinstance(i, Image)

    from willow.image import ImageFile, JPEGImageFile
    isinstance(i, ImageFile)
    isinstance(i, JPEGImageFile)
```

If it succeeded, this will return a subclass of `ImageFile` (which itself is a subclass of *`Image`*).

The `ImageFile` subclass it chooses depends on the format of the image (detected by inspecting the file header). In this case, it used `JPEGImageFile` as the image we loaded was in JPEG format.

Using different image classes for different formats allows Willow to decide which plugin to use for performing operations on the image. For example, Willow will always favour Wand for resizing GIF images but will always favour Pillow for resizing JPEG and PNG images.

From an existing Pillow object

You can create a Willow `Image` from an existing `PIL.Image` object by creating an instance of the `PillowImage` class (passing the `PIL.Image` object as the only parameter):

```
from willow.plugins.pillow import PillowImage

pillow_image = PIL.Image.open(...)

i = PillowImage(pillow_image)

isinstance(i, PillowImage)

from willow.image import Image
isinstance(i, Image)
```

The same can be done with Wand and OpenCV, which use the `WandImage` and `OpenCVColorImage` classes respectively.

Basic image operations

Here's where Willow gets fancy, all operations in all plugins are available as methods on every image. If an operation is called but doesn't exist in the image's current class, a conversion will be performed under the hood.

Willow will do it's best to maintain the quality of the image, it'll decide how to convert based on the images format and whether it has animation or transparency. However it is not always easy

This means you can focus on making the code look clear and leave Willow to choose which plugin is best to perform an operation.

Getting the image size

You can call the `get_size()` method which returns the width and height as a tuple of two integers:

```
# For example, 'i' is a 200x200 pixel image
i.get_size() == (200, 200)
```

For animated GIFs, you can get the number of frames by calling the `Image.get_frame_count()` method:

```
i.get_frame_count() == 34
```

Resizing images

To resize an image, call the `resize()` method. This stretches the image to fit the new size.

It takes a single argument, a two element sequence of integers containing the width and height of the final image.

It returns a new *Image* object containing the resized image. The original image is not modified.

```
i = i.resize((100, 100))

isinstance(i, Image)
i.get_size() == (100, 100)
```

Rotating images

To rotate an image, call the `rotate()` method. This rotates the image clockwise, by a multiple of 90 degrees (i.e 90, 180, 270).

It returns a new *Image* object containing the rotated image. The original image is not modified.

Cropping images

To crop an image, call the `crop()` method. This cuts the specified rectangle from the source image.

It takes a single argument, a four element sequence of integers containing the location of the left, top, right and bottom edges to cut out.

It returns a new *Image* object containing the cropped region. The original image is not modified.

```
i = i.crop((100, 100, 300, 300))

isinstance(i, Image)
i.get_size() == (200, 200)
```

Setting a background colour

If the image has transparency, you can replace the transparency with a solid background colour using the `set_background_color_rgb()` method.

It takes the background color as a three element tuple of integers between 0 - 255 (representing the red, green and blue channels respectively).

It returns a new *Image* object containing the background color and the alpha channel removed. The original image is not modified.

```
# Sets background color to white
i = i.set_background_color_rgb((255, 255, 255))

isinstance(i, Image)
i.has_alpha() == False
```

Detecting features

Feature detection in Willow is provided by OpenCV so make sure it's installed first.

To detect features in an image, use the `detect_features()` operation. This will return a list of tuples, containing the x and y coordinates of each feature that was detected in the image.

```
features = i.detect_features()

features == [
    (12, 53),
    (74, 44),
    ...
]
```

Under the hood, this uses OpenCV's GoodFeaturesToTrack function that finds the prominent corners in the image.

Detecting faces

Face detection in Willow is provided by OpenCV so make sure it's installed first.

To detect features in an image, use the `detect_faces()` operation. This will return a list of tuples, containing the left, top, right and bottom positions in the image where each face appears.

```
faces = i.detect_faces()

faces == [
    (12, 53, 65, 102),
    (1, 44, 74, 93),
    ...
]
```

Under the hood, this uses OpenCV's HaarDetectObjects function that performs Haar cascade classification on the image. The default cascade file that gets used is `haarcascade_frontalface_alt2` from OpenCV, but this can be changed by setting the `cascade_filename` keyword argument to an absolute path pointing to the file:

```
import os

faces = i.detect_faces(cascade_filename=os.abspath('cascades/my_cascade_file.xml'))

faces == [
    (12, 53, 65, 102),
    (1, 44, 74, 93),
    ...
]
```

Saving images

In Willow there are separate save operations for each image format:

- `save_as_jpeg()`
- `save_as_png()`
- `save_as_gif()`
- `save_as_webp()`

All three take one positional argument, the file-like object to write the image data to.

For example, to save an image as a PNG file:

```
with open('out.png', 'wb') as f:
    i.save_as_png(f)
```

Changing the quality setting

`save_as_jpeg()` and `save_as_webp()` takes a `quality` keyword argument, which is a number between 1 and 100. It defaults to 85 for `save_as_jpeg()` and 80 for `save_as_webp()`. Decreasing this number will decrease the output file size at the cost of losing image quality.

For example, to save an image with low quality:

```
with open('low_quality.jpg', 'wb') as f:
    i.save_as_jpeg(f, quality=40)
```

Progressive JPEGs

By default, JPEG's are saved in the same format as their source file but you can force Willow to always save a "progressive" JPEG file by setting the `progressive` keyword argument to `True`:

```
with open('progressive.jpg', 'wb') as f:
    i.save_as_jpeg(f, progressive=True)
```

Lossless WebP

You can encode the image to WebP without any loss by setting the `lossless` keyword argument to `True`:

```
with open('lossless.webp', 'wb') as f:
    i.save_as_webp(f, lossless=True)
```

Image optimisation

`save_as_jpeg()` and `save_as_png()` both take an `optimize` keyword that when set to true, will output an optimized image.

```
with open('optimized.jpg', 'wb') as f:
    i.save_as_jpeg(f, optimize=True)
```

This feature is currently only supported in the Pillow backend, if you use Wand this argument will be ignored.

Extending Willow

This section describes how to extend Willow with custom operations, image formats and plugins.

Don't forget to look at the *concepts* section first!

Implementing new operations

You can add operations to any existing image class and register them by calling the `Registry.register_operation()` method passing it the image class, name of the operation and the function to call when the operation is used.

For example, let's implement a `blur` operation for both the `PillowImage` and `WandImage` classes:

```
from willow.registry import registry
from willow.plugins.pillow import PillowImage
from willow.plugins.wand import WandImage

def pillow_blur(image):
    from PIL import ImageFilter

    blurred_image = image.image.filter(ImageFilter.BLUR)
    return PillowImage(blurred_image)

def wand_blur(image):
    # Wand modifies images in place so clone it first to prevent
    # altering the original image
    blurred_image = image.image.clone()
    blurred_image.gaussian_blur()
    return WandImage(blurred_image)


# Register the operations in Willow

registry.register_operation(PillowImage, 'blur', pillow_blur)
registry.register_operation(WandImage, 'blur', wand_blur)
```

It is not required to support both `PillowImage` and `WandImage` but it's recommended that libraries support both for maximum compatibility. You must support Wand if you need animated GIF support.

Implementing custom image classes

You can create your own image classes and register them by calling the `Registry.register_image_class()` method. All image classes must be a subclass of `willow.image.Image`.

Methods on image classes can be decorated with `@Image.operation`, `@Image.converter_from` or `@Image.converter_to` which will make Willow automatically register those methods as operations or converters.

For example, let's implement our own image class for Pillow:

```
from __future__ import absolute_import

import PIL.Image

from willow.image import (
    Image,
```

(continues on next page)

(continued from previous page)

```
    JPEGImageFile,
    PNGImageFile,
    GIFImageFile,
)


class NewPillowImage(Image):
    def __init__(self, image):
        self.image = image


    # Informational operations

    @Image.operation
    def get_size(self):
        return self.image.size

    @Image.operation
    def has_alpha(self):
        img = self.image
        return img.mode in ('RGBA', 'LA') or (img.mode == 'P' and 'transparency' in␣
↪img.info)

    @Image.operation
    def has_animation(self):
        # Animation is not supported by PIL
        return False


    # Resize and crop operations

    @Image.operation
    def resize(self, size):
        return PillowImage(image.resize(size, PIL.Image.ANTIALIAS))

    @Image.operation
    def crop(self, rect):
        return PillowImage(self.image.crop(rect))


    # Converter from supported file formats, this is where the image is opened

    # Pillow doesn't support GIFs very well. Adding a cost will make Willow try
    # a different image class first. The default cost for all converters is 100.

    @classmethod
    @Image.converter_from(JPEGImageFile)
    @Image.converter_from(PNGImageFile)
    @Image.converter_from(GIFImageFile, cost=200)
    @Image.converter_from(BMPImageFile)
    def open(cls, image_file):
        image_file.f.seek(0)
        image = PIL.Image.open(image_file.f)

        return cls(image)
```

The image class can then be registered by calling `Registry.register_image_class()`:

```
from willow.registry import registry

from newpillow import NewPillowImage

registry.register_image_class(NewPillowImage)
```

This will also register all operations and converters defined on the class.

Plugins

Plugins allow multiple image classes and/or operations to be registered together. They are Python modules with any of the following attributes defined: `willow_image_classes`, `willow_operations` or `willow_converters`.

For example, we can convert the Python module in the example above into a Willow plugin by adding the following line at the bottom of the file:

```
willow_image_classes = [NewPillowImage]
```

It can now be registered using the `Registry.register_plugin()` method:

```
from willow.registry import registry

import newpillow

registry.register_plugin(newpillow)
```

1.4 Reference

1.4.1 The `Image` class

class Image

classmethod open(*file*)
Opens the provided image file detects the format from the image header using Python's `imghdr` module.

Returns a subclass of `ImageFile`

If the image format is unrecognised, this throws a `willow.image.UnrecognisedImageFormatError` (a subclass of `IOError`)

classmethod operation()
A decorator for registering operations.

The operations will be automatically registered when the image class is registered.

```
from willow.image import Image

class MyImage(Image):

    @Image.operation
    def resize(self, size):
        return MyImage(self.image.resize(size))
```

classmethod converter_from(*other_classes*, *cost=100*)

A decorator for registering a "from" converter, which is a classmethod that converts an instance of another image class into an instance of this one.

The `other_classes` parameter specifies which classes this converter can convert from. It can be a single class or a list.

```
from willow.image import Image

class MyImage(Image):
    ...

    @classmethod
    @Image.converter_from(JPEGImageFile)
    def open_jpeg_file(cls, image_file):
        return cls(image=open_jpeg(image_file.f))
```

It can also be applied multiple times to the same function allowing different costs to be specified for different classes:

```
@classmethod
@Image.converter_from([JPEGImageFile, PNGImageFile])
@Image.converter_from(GIFImageFile, cost=200)
def open_file(cls, image_file):
    ...
```

classmethod converter_to(*other_class*, *cost=100*)

A decorator for registering a "to" converter, which is a method that converts this image into an instance of another class.

The `other_class` parameter specifies which class this function converts to. An individual "to" converter can only convert to a single class.

```
from willow.image import Image

class MyImage(Image):
    ...

    @Image.converter_to(PillowImage)
    def convert_to_pillow(self):
        image = PIL.Image()  # Code to create PIL image object here
        return PillowImage(image)
```

1.4.2 Builtin operations

Here's a full list of operations provided by Willow out of the box:

get_size()

Returns the size of the image as a tuple of two integers:

```
width, height = image.get_size()
```

get_frame_count()

Returns the number of frames in an animated image:

```
number_of_frames = image.get_frame_count()
```

has_alpha()
Returns `True` if the image has an alpha channel.

```
if image.has_alpha():
    # Image has alpha
```

has_animation()
Returns `True` if the image is animated.

```
if image.has_animation():
    # Image has animation
```

resize(*size*)
(Pillow/Wand only)

Stretches the image to fit the specified size. Size must be a sequence of two integers:

```
# Resize the image to 100x100 pixels
resized_image = source_image.resize((100, 100))
```

crop(*region*)
(Pillow/Wand only)

Cuts out the specified region of the image. The region must be a sequence of four integers (top, left, right, bottom):

```
# Cut out a square from the middle of the image
cropped_image = source_image.resize((100, 100, 200, 200))
```

set_background_color_rgb(*color*)
(Pillow/Wand only)

If the image has an alpha channel, this will add a background colour using the alpha channel as a mask. The alpha channel will be removed from the resulting image.

The background colour must be specified as a tuple of three integers with values between 0 - 255.

This operation will convert the image to RGB format, but will not do anything if there is not an alpha channel.

```
# Set the background colour of the image to white
image = source_image.set_background_color_rgb((255, 255, 255))
```

auto_orient()
(Pillow/Wand only)

Some JPEG files have orientation data in an EXIF tag that needs to be applied to the image. This method applies this orientation to the image (it is a no-op for other image formats).

This should be run before performing any other image operations.

```
image = image.auto_orient()
```

detect_features()
(OpenCV only)

Uses OpenCV to find the most prominent corners in the image. Useful for detecting interesting features for cropping against.

Returns a list of two integer tuples containing the coordinates of each point on the image

```
points = image.detect_features()
```

detect_faces(*cascade_filename*)
(OpenCV only)

Uses OpenCV's cascade classification to detect faces in the image.

By default the `haarcascade_frontalface_alt2.xml` (provided by OpenCV) cascade file is used. You can specifiy the filename to a different cascade file in the first parameter.

Returns a list of four integer tuples containing the left, top, right, bottom locations of each face detected in the image.

```
faces = image.detect_faces()
```

save_as_jpeg(*file*, *quality=85*, *optimize=False*)
(Pillow/Wand only)

Saves the image to the specified file-like object in JPEG format.

Note: If the image has an alpha channel, this operation may raise an exception or save a broken image (depending on the backend being used). To resolve this, use the `set_background_color_rgb()` method to replace the alpha channel with a solid background color before saving as JPEG.

Returns a `JPEGImageFile` wrapping the file.

```
with open('out.jpg', 'wb') as f:
    image.save_as_jpeg(f)
```

save_as_png(*file*, *optimize=False*)
(Pillow/Wand only)

Saves the image to the specified file-like object in PNG format.

Returns a `PNGImageFile` wrapping the file.

```
with open('out.png', 'wb') as f:
    image.save_as_png(f)
```

save_as_gif(*file*)
(Pillow/Wand only)

Saves the image to the specified file-like object in GIF format.

returns a `GIFImageFile` wrapping the file.

```
with open('out.gif', 'wb') as f:
    image.save_as_gif(f)
```

save_as_webp(*file*, *quality=80*, *lossless=False*)
(Pillow/Wand only)

Saves the image to the specified file-like object in WEBP format.

returns a `WebPImageFile` wrapping the file.

```
with open('out.webp', 'wb') as f:
    image.save_as_webp(f)
```

get_pillow_image()
(Pillow only)

Returns a `PIL.Image` object for the specified image. This may be useful for reusing existing code that requires a Pillow image.

```
do_thing(image.get_pillow_image())
```

You can convert a `PIL.Image` object back into a Willow *Image* using the `PillowImage` class:

```
import PIL.Image
from willow.plugins.pillow import PillowImage

pillow_image = PIL.Image.open('test.jpg')
image = PillowImage(pillow_image)

# Now you can use any Willow operation on that image
faces = image.detect_faces()
```

get_wand_image()
(Wand only)

Returns a `Wand.Image` object for the specified image. This may be useful for reusing existing code that requires a Wand image.

```
do_thing(image.get_wand_image())
```

You can convert a `Wand.Image` object back into a Willow *Image* using the `WandImage` class:

```
from wand.image import Image
from willow.plugins.wand import WandImage

# wand_image is an instance of Wand.Image
wand_image = Image(filename='pikachu.png')
image = WandImage(wand_image)

# Now you can use any Willow operation on that image
faces = image.detect_faces()
```

1.5 Changelog

1.5.1 1.4 (26/05/2020)

- Implemented save quality/lossless options for WebP (@mozgsml)
- Added missing docs for WebP support (@mozgsml)

1.5.2 1.3 (16/10/2019)

- Added `.get_frame_count()` operaton (@kaedroho)

1.5.3 1.2 (11/10/2019)

- Added WebP support (@frmdstryr)
- Added `.rotate()` operaton (@mrchrisadams & @simo97)

1.5.4 1.1 (04/12/2017)

- Added *set_background_color_rgb* operation
- Update MANIFEST.in (Sanny Kumar)

1.5.5 1.0 (04/08/2017)

- OpenCV 3 support (Will Giddens)
- Removed Apple copyrighted ICC profile from orientation test images (Christopher Hoskin)
- Fix: Altered *detect_features* in OpenCV 3 to return a list instead of a numpy array (Trent Holliday)
- Support for TIFF files (Maik Hoepfel)
- Support for BMP files was made official (Maik Hoepfel)

1.5.6 0.4 (05/10/2016)

- Support for image optimisation and saving progressive JPEG files
- Added documentation

1.5.7 0.3.1 (16/05/2016)

- Fixed crash in the Pillow auto_orient operation when the image has an invalid Orientation EXIF Tag (Sigurdur J Eggertsson)
- The `auto_orient` operation now catches all errors raised while reading EXIF data (Tomas Olander)
- Palette formatted PNG and GIF files that have transparency no longer lose their transparency when resizing them

1.5.8 0.3 (09/03/2016)

A major internals refactor has taken place in this release, there are a number of breaking changes:

- The Image class is now immutable. Previously, "resize" and "crop" operations altered the image in-place but now they now always return a new image leaving the original untouched.
- There are now multiple Image classes. Each one represents possible state the image can be in (eg, in a file, loaded in Pillow, etc). Operations can return an image in a different class to what the operation was performed on.
- The "backends" have been renamed to "plugins".
- A new registry module has been added which can be used for registering new plugins and operations.
- The "original_format" attribute has been deprecated.

Other changes in this release:

- Added auto_orient operation

1.5.9 0.2.1 (27/05/2015)

- JPEGs are now detected from first two bytes of their file. Allowing non JFIF/EXIF JPEG images to be loaded

1.5.10 0.2 (01/04/2015)

- Added loader for BMP files
- Added has_alpha and has_animation operations
- Added get_pillow_image and get_wand_image operations
- Added save_as_{jpeg,png,gif} operations
- Crop and resize now all arguments in a tuple (Similar to Pillow)
- Dropped Python 2.6 and 3.2 support
- Formats now detected using images header instead of extension
- Now possible to specify alternative cascade file for face detection
- Fix: Images now saved in the same format they were loaded
- Fix: 1 and P formatted images now converted to RGB when saving to JPEG

1.5.11 0.1 (22/02/2015)

Initial release

Index

A

auto_orient(), 13

C

converter_from(), 11
converter_to(), 12
crop(), 13

D

detect_faces(), 14
detect_features(), 13

G

get_frame_count(), 12
get_pillow_image(), 14
get_size(), 12
get_wand_image(), 15

H

has_alpha(), 12
has_animation(), 13

I

Image (*built-in class*), 11

O

open(), 11
operation(), 11

R

resize(), 13

S

save_as_gif(), 14
save_as_jpeg(), 14
save_as_png(), 14
save_as_webp(), 14
set_background_color_rgb(), 13