ril

Release 0.4.0

Cryptex

Sep 11, 2022

CONTENTS

Rust Imaging Library's Python binding: A performant and high-level image processing library for Python written in Rust

CHAPTER

ONE

RIL

CHAPTER

TWO

CONTENTS

2.1 Ril API Reference

Ril provide a performant and high-level image processing library for Python written in Rust.

2.1.1 Image

class ril.**Image**

A high-level image representation.

This represents a static, single-frame image. See *ImageSequence* for information on opening animated or multi-frame images.

bands()

Return the bands of the image.

Return type
Tuple[*L*, ...]

Raises
TypeError – The image is not of mode *RGB* or *RGBA*.

crop(*x1*, *y1*, *x2*, *y2*)

Crops this image in place to the given bounding box.

Parameters

- **x1** (*int*) – The x axis of the upper-left corner
- **y1** (*int*) – The y axis of the upper-left corner
- **x2** (*int*) – The x axis of the lower-right corner
- **y2** (*int*) – The y axis of the lower-right corner

dimensions

Returns the dimensions of the image.

Type
Tuple[int, int]

draw(*entity*)

Draws an object or shape onto this image.

Parameters
entity (Union[*Rectangle*, *Ellipse*]) – The entity to draw on the image.

encode(*encoding*)

Encodes the image with the given encoding and returns *bytes*.

Parameters
encoding (*str*) – The encoding of the image.

Returns
The encoded bytes of the image.

Return type
bytes

Raises

- **ValueError** – The encoding is invalid.
- **RuntimeError** – Failed to encode the image.

flip()

Flips this image vertically (about the x-axis) in place.

format

Returns the encoding format of the image.

Note: This is nothing more but metadata about the image. When saving the image, you will still have to explicitly specify the encoding format.

Type
str

from_bands(**bands*)

Creates a new image from the given bands.

Parameters
bands (* *L*) – The bands of the image.

from_bytes(*bytes*, *format=None*)

Decodes an image with the explicitly given image encoding from the raw bytes.

if *format* is not provided then it will try to infer its encoding.

Parameters

- **bytes** (*bytes*) – The bytes of the Image.
- **format** (*Optional[str], default: None*) – The format of the image, defaults to *None*.

Raises

- **ValueError** – Raised if the format provided is invalid.
- **RuntimeError** – Raised if the image can't be decoded or the format is unknown.

from_pixels(*width*, *pixels*)

Creates a new image shaped with the given width and a 1-dimensional sequence of pixels which will be shaped according to the width.

Parameters

- **width** (*int*) – The width of the image.

- **pixels** (List[*Pixel*]) – A List of pixels.

get_pixel(*x*, *y*)

Returns the pixel at the given coordinates.

Parameters

- **x** (*int*) – The x axis
- **y** (*int*) – The y axis

Returns
The pixel of that specific coordinate.

Return type
Union[*BitPixel*, *L*, *Rgb*, *Rgba*]

height

Returns the height of the image.

Type
int

invert()

Inverts the image in-place.

mask_alpha(*mask*)

Masks the alpha values of this image with the luminance values of the given single-channel L image.

If you want to mask using the alpha values of the image instead of providing an L image, you can split the bands of the image and extract the alpha band.

This masking image must have the same dimensions as this image.

Parameters
mask (*Image*) – The mask to use

Raises
ValueError – The mask provided is not of mode *L*

mirror()

Mirrors, or flips this image horizontally (about the y-axis) in place.

mode

Returns the mode of the image.

Type
str

new(*width*, *height*, *fill*)

Creates a new image with the given width and height, with all pixels being set intially to *fill*.

Parameters

- **width** (*int*) – The width of the Image.
- **height** (*int*) – The height of the Image.
- **fill** (*Pixel*) – The pixel used to fill the image.

Examples

```
Image.new(100, 100, Pixel.from_rgb(255, 255, 255))
```

open(*path*)

Opens a file from the given path and decodes it into an image.

The encoding of the image is automatically inferred. You can explicitly pass in an encoding by using the *from_bytes()* method.

Parameters
path (*str*) – The path to the image.

Raises

- **ValueError** – The file extension is invalid.
- **RuntimeError** – Failed to infer file format or Failed to decode image.

overlay_mode

Returns the overlay mode of the image.

Type
OverlayMode

paste(*x*, *y*, *image*, *mask=None*)

Pastes the given image onto this image at the given x and y axis.

If *mask* is provided it will be masked with the given masking image.

Currently, only BitPixel images are supported for the masking image.

Parameters

- **x** (*int*) – The x axis
- **y** (*int*) – The y axis
- **image** (*Image*) – The image to paste.
- **mask** (Optional[*Image*], default: None) – The mask to use, defaults to *None*

Raises
ValueError – The mask provided is not of mode *BitPixel*

pixels()

Returns a 2D list representing the pixels of the image. Each list in the list is a row.

For example:

[[Pixel, Pixel, Pixel], [Pixel, Pixel, Pixel]]

where the width of the inner list is determined by the width of the image.

> **Warning: This function involves heavy operation**
>
> This function requires multiple iterations, so it is a heavy operation for larger image.

Returns
The pixels of the image.

Return type
List[List[Union[*BitPixel*, *L*, *Rgb*, *Rgba*]]]

resize(*width*, *height*, *algorithm*)

Resizes this image in place to the given dimensions using the given resizing algorithm in place.

Parameters

- **width** (*int*) – The target width to resize to
- **height** (*int*) – The target height to resize to
- **algorithm** (*ResizeAlgorithm*) – The resize algorithm to use

save(*path*, *encoding=None*)

Saves the image to the given path. If encoding is not provided, it will attempt to infer it by the path/filename's extension You can try saving to a memory buffer by using the *encode()* method.

Parameters

- **path** (*str*) – The path to save the image to.
- **encoding** (*Optional[str], default: None*) – The encoding of the image, defaults to *None*.

Raises

- **ValueError** – The encoding provided is invalid.
- **RuntimeError** – Failed to encode the image or Failed to infer the image format.

set_pixel(*x*, *y*, *pixel*)

Sets the pixel at the given coordinates to the given pixel.

Parameters

- **x** (*int*) – The x axis
- **y** (*int*) – The y axis
- **pixel** (*Pixel*) – The pixel to set it to

width

Returns the width of the image.

Type
int

2.1.2 Pixel

There are two pixel types.

Pixel and other pixel classes.

Pixel is what the user creates, to represent the pixel type they desire.

Other pixel types are usually returned from the library.

This is done due to some limitation between converting types.

class ril.**BitPixel**

Represents a single-bit pixel that represents either a pixel that is on or off.

value

Whether the pixel is on.

Type
bool

class ril.**L**

Represents an L, or luminance pixel that is stored as only one single number representing how bright, or intense, the pixel is.

This can be thought of as the "unit channel" as this represents only a single channel in which other pixel types can be composed of.

value

The luminance value of the pixel, between 0 and 255.

Type
int

class ril.**Rgb**

Represents an RGB pixel.

b

The blue component of the pixel.

Type
int

g

The green component of the pixel.

Type
int

r

The red component of the pixel.

Type
int

class ril.**Rgba**

Represents an RGBA pixel.

a

The alpha component of the pixel.

Type
int

b

The blue component of the pixel.

Type
int

g

The green component of the pixel.

Type
int

r

The red component of the pixel.

Type
int

class ril.**Pixel**

The user created Pixel type.

from_bitpixel(*value*)

Create a bitpixel.

Parameters
value (*bool*) – Whether the pixel is on.

from_l(*value*)

Create a L Pixel.

Parameters
value (*int*) – The luminance value of the pixel, between 0 and 255.

from_rgb(*r*, *g*, *b*)

Creates a Rgb Pixel

Parameters

- **r** (*int*) – The red component of the pixel.
- **g** (*int*) – The green component of the pixel.
- **b** (*int*) – The blue component of the pixel.

from_rgba(*r*, *g*, *b*, *a*)

Creates a Rgba Pixel

Parameters

- **r** (*int*) – The red component of the pixel.
- **g** (*int*) – The green component of the pixel.
- **b** (*int*) – The blue component of the pixel.
- **a** (*int*) – The alpha component of the pixel.

2.1.3 Draw

class ril.**Border**(*color*, *thickness*, *position*)

Represents a shape border.

Parameters

- **color** (*Pixel*) – The color of the border
- **thickness** (*int*) – The thickness of the border
- **position** (*str*) – The position of the border

Raises
ValueError – The position is not one of *inset*, *center*, or *outset*

color

The color of the border.

Type
Pixel

position

The position of the border.

Type
str

thickness

The thickness of the border, in pixels.

Type
int

class ril.**Rectangle**(*, *position*, *size*, *border*, *fill*, *overlay*)

A rectangle.

> **Warning:** Using any of the predefined construction methods will automatically set the position to (0, 0). If you want to specify a different position, you must set the position with *.position*
>
> You must specify a width and height for the rectangle with something such as with_size. If you don't, a panic will be raised during drawing. You can also try using from_bounding_box to create a rectangle from a bounding box, which automatically fills in the size.
>
> Additionally, a panic will be raised during drawing if you do not specify either a fill color or a border. these can be set with *.fill* and *.border* respectively.

Parameters

- **position** (*Tuple[int, int]*) – The position of the rectangle
- **size** (*Tuple[int, int]*) – The size of the rectangle
- **border** (Optional[*Border*]) – The border of the ellipse.
- **fill** (Optional[*Pixel*]) – The color to use for filling the rectangle
- **overlay** (Optional[*OverlayMode*]) – The overlay mode of the rectangle.

Raises
ValueError – The overlay mode provided is not one of *replace*, or *merge*

border

The border of the rectangle, or None if there is no border.

Type
Border

fill

The color used to fill the rectangle.

Type
Optional[Union[*BitPixel*, *L*, *Rgb*, *Rgba*]]

from_bounding_box(*x1*, *y1*, *x2*, *y2*)

Creates a new rectangle from two coordinates specified as 4 parameters. The first coordinate is the top-left corner of the rectangle, and the second coordinate is the bottom-right corner of the rectangle.

Parameters

- **x1** (*int*) – The x axis of the upper-left corner
- **y1** (*int*) – The y axis of the upper-left corner
- **x2** (*int*) – The x axis of the lower-right corner
- **y2** (*int*) – The y axis of the lower-right corner

overlay

The overlay mode of the rectangle.

Type
Optional[*OverlayMode*]

position

The position of the rectangle. The top-left corner of the rectangle will be rendered at this position.

Type
Tuple[int, int]

size

The dimensions of the rectangle, in pixels.

Type
Tuple[int, int]

class ril.**Ellipse**(*, *position*, *radii*, *border*, *fill*, *overlay*)

An ellipse, which could be a circle.

> **Warning:** Using any of the predefined constructors will automatically set the position to (0, 0) and you must explicitly set the size of the ellipse with *.size* in order to set a size for the ellipse. A size must be set before drawing.
>
> This also does not set any border or fill for the ellipse, you must explicitly set either one of them.

Parameters

- **position** (*Tuple[int, int]*) – The position of the ellipse
- **radii** (*Tuple[int, int]*) – The radii of the ellipse
- **border** (Optional[*Border*]) – The border of the ellipse.
- **fill** (Optional[*Pixel*]) – The color to use for filling the ellipse
- **overlay** (*Optional[str]*) – The overlay mode of the ellipse.

border

The border of the ellipse.

Type
Optional[*Border*]

circle(*x*, *y*, *radius*)

Creates a new circle with the given center position and radius.

Parameters

- **x** (*int*) – The x axis
- **y** (*int*) – The y axis
- **radius** (*int*) – The radius

fill

The color used to fill the ellipse.

Type
Optional[Union[*BitPixel*, *L*, *Rgb*, *Rgba*]]

from_bounding_box(*x1*, *y1*, *x2*, *y2*)

Creates a new ellipse from the given bounding box.

Parameters

- **x1** (*int*) – The x axis of the upper-left corner
- **y1** (*int*) – The y axis of the upper-left corner
- **x2** (*int*) – The x axis of the lower-right corner
- **y2** (*int*) – The y axis of the lower-right corner

Return type
Ellipse

overlay

The overlay mode of the ellipse.

Type
Optional[*OverlayMode*]

position

The center position of the ellipse. The center of this ellipse will be rendered at this position.

Type
Tuple[int, int]

radii

The radii of the ellipse, in pixels; (horizontal, vertical).

Type
Tuple[int, int]

2.1.4 Sequence

class ril.**ImageSequence**

Represents a sequence of image frames such as an animated image.

See *Image* for the static image counterpart, and see *Frame* to see how each frame is represented in an image sequence.

The iterator is exhausive, so when you iterate through *ImageSequence* like

```
seq = ImageSequence.from_bytes(bytes)
list(seq) # [...]
# But if you do it again
list(seq) # []
# It will return a empty list
```

Note: Any change made to the *Frame* will not be reflected to the *ImageSequence*, so you must create a new *ImageSequence* after you make changes to the frames.

encode()

Encodes the image with the given encoding and returns *bytes*.

Parameters
encoding (*str*) – The encoding to encode to.

Returns
The encoded bytes.

Return type
bytes

from_bytes(*bytes*, *format*)

Decodes a sequence with the explicitly given image encoding from the raw bytes.

if *format* is not provided then it will try to infer its encoding.

Parameters

- **bytes** (*bytes*) – The bytes of the image.
- **format** (*Optional[str], default: None*) – The format of the image.

Raises

- **ValueError** – The format provided is invalid.
- **RuntimeError** – Failed to decode the image or Failed to infer the image's format.

from_frames()

Creates a new image sequence from the given frames

Parameters
frames (List[*Frame*]) – The list of frames to create the sequence from

open(*path*)

Opens a file from the given path and decodes it into an *ImageSequence*.

The encoding of the image is automatically inferred. You can explicitly pass in an encoding by using the *from_bytes()* method.

Parameters
path (*str*) – The path to the image.

Raises

- **ValueError** – The file extension is invalid.
- **RuntimeError** – Failed to infer file format or Failed to decode image.

save()

Saves the image to the given path. If encoding is not provided, it will attempt to infer it by the path/filename's extension You can try saving to a memory buffer by using the *encode()* method.

Parameters
path (*str*) – The path to the image.

Raises

- **ValueError** – The file extension is invalid.
- **RuntimeError** – Failed to infer file format or Failed to decode image.

class ril.**Frame**(*image*)

Represents a frame in an image sequence. It encloses *Image* and extra metadata about the frame.

Parameters
image (*Image*) – The image used for this frame.

delay

Returns the delay duration for this frame.

Type
int

dimensions

Returns the dimensions of this frame.

Type
Tuple[int, int]

disposal

Returns the disposal method for this frame.

Type
DisposalMethod

image

Returns the image this frame contains.

Type
Image

2.1.5 Text

class ril.**Font**

Represents a single font along with its alternatives used to render text. Currently, this supports TrueType and OpenType fonts.

from_bytes(*bytes*, *optimal_size*)

Loads the font from the given bytes.

Note: The optimal size is not the fixed size of the font - rather it is the size to optimize rasterizing the font for.

Lower sizes will look worse but perform faster, while higher sizes will look better but perform slower. It is best to set this to the size that will likely be the most use

---

Parameters

- **path** (*str*) – The path of the font.
- **optimal_size** (*float*) – The optimal size of the font.

Raises

- **IOError** – Fails to read the font file.
- **RuntimeError** – Fails to load the font.

open(*path*, *optimal_size*)

Opens the font from the given path.

Note: The optimal size is not the fixed size of the font - rather it is the size to optimize rasterizing the font for.

Lower sizes will look worse but perform faster, while higher sizes will look better but perform slower. It is best to set this to the size that will likely be the most use

---

Parameters

- **path** (*str*) – The path of the font.
- **optimal_size** (*float*) – The optimal size of the font.

Raises

- **IOError** – Fails to read the font file.
- **RuntimeError** – Fails to load the font.

See also:

from_bytes()

optimal_size

Returns the optimal size, in pixels, of this font.

Note: The optimal size is not the fixed size of the font - rather it is the size to optimize rasterizing the font for.

Lower sizes will look worse but perform faster, while higher sizes will look better but perform slower. It is best to set this to the size that will likely be the most used.

Type
float

class ril.**TextSegment**

Represents a text segment that can be drawn.

See *TextLayout* for a more robust implementation that supports rendering text with multiple styles. This type is for more simple and lightweight usages.

Additionally, accessing metrics such as the width and height of the text cannot be done here, but can be done in TextLayout since it keeps a running copy of the layout. Use TextLayout if you will be needing to calculate the width and height of the text. Additionally, TextLayout supports text anchoring, which can be used to align text.

If you need none of these features, *TextSegment* should be used in favor of being much more lightweight.

Parameters

- **font** (*Font*) – The font to use to render the text.
- **text** (*str*) – The text to render.
- **fill** (*Pixel*) – The fill color the text will be in.
- **position** (*Optional[Tuple[int, int]]*) – The position the text will be rendered at.

  This must be set before adding any text segments!

  Either with *position* or by passing it to the constructor.
- **size** (*Optional[float]*) – The size of the text in pixels.
- **overlay** (Optional[*OverlayMode*]) – The overlay mode to use when rendering the text.
- **width** (*Optional[int]*) – The width of the text layout.
- **wrap** (Optional[*WrapStyle*]) – The wrapping style of the text. Note that text will only wrap if *width* is set. If this is used in a *TextLayout*, this is ignored and `WrapStyle.Wrap` is used instead.

Warning: As this class contains the data of an entire font, copying this class is expensive.

`fill`

The fill color of the text segment.

Type
List[List[Union[*BitPixel*, *L*, *Rgb*, *Rgba*]]]

`font`

The font of the text segment.

Warning: Due to design limitation, accessing font requires a deep clone each time, which is expensive.

Type
Font

`overlay`

The overlay mode of the text segment.

Type
OverlayMode

`position`

The position of the text segment.

Type
Tuple[int, int]

`size`

The size of the text segment in pixels.

Type
float

text

The content of the text segment.

Type
str

width

The width of the text box.

> **Warning:** If this is used in a *TextLayout*, this is ignored and *TextLayout.width()* is used instead.

Type
float

wrap

The wrapping style of the text segment.

Type
WrapStyle

class ril.**TextLayout**(*font*, *text*, *fill*, *position=None*, *size=None*, *overlay=None*, *width=None*, *wrap=None*)

Represents a high-level text layout that can layout text segments, maybe with different fonts.

This is a high-level layout that can be used to layout text segments. It can be used to layout text segments with different fonts and styles, and has many features over *TextSegment* such as text anchoring, which can be useful for text alignment. This also keeps track of font metrics, meaning that unlike *TextSegment*, this can be used to determine the width and height of text before rendering it.

This is less efficient than *TextSegment* and you should use *TextSegment* if you don't need any of the features TextLayout provides.

Parameters

- **position** (*Optional[Tuple[int, int]]*) – The position the text will be rendered at.

  This must be set before adding any text segments!

  Either with *position* or by passing it to the constructor.
- **horizontal_anchor** (Optional[*HorizontalAnchor*]) – The horizontal anchor of the text.
- **vertical_anchor** (Optional[*VerticalAnchor*]) – The vertical anchor of the text.
- **wrap** (Optional[*WrapStyle*]) – Sets the wrapping style of the text. Make sure to also set the wrapping width using *width* for wrapping to work.

  This must be set before adding any text segments!

> **Warning:** As this class contains the data of one or more font(s), copying this class can be extremely expensive.

bounding_box

Returns the bounding box of the text. Left and top bounds are inclusive; right and bottom bounds are exclusive.

Type
Tuple[int, int, int, int]

`centered`()

Sets the horizontal anchor and vertial anchor of the text to be centered. This makes the position of the text be the center as opposed to the top-left corner.

`dimensions`

Returns the width and height of the text.

> **Warning:** This is a slightly expensive operation and is not a simple getter.

> **Note:** If you want both width and height, use *dimensions*.

Type
Tuple[int, int]

`height`

Returns the height of the text.

> **Warning:** This is a slightly expensive operation and is not a simple getter.

> **Note:** If you want both width and height, use *dimensions*.

Type
int

`horizontal_anchor`

Sets the horizontal anchor of the text layout.

`position`

Sets the position of the text layout.

This must be set before adding any text segments!

`push_basic_text`(*font*, *text*, *fill*)

Pushes a basic text to the text layout. Adds basic text to the text layout. This is a convenience method that creates a *TextSegment* with the given font, text, and fill and adds it to the text layout. The size of the text is determined by the font's optimal size.

Parameters

- **font** (*Font*) – The font to use for the text.
- **text** (*str*) – The text to add.
- **fill** (*Pixel*) – The color of the text.

`push_segment`(*segment*)

Pushes a text segment to the text layout.

Parameters
segment (*TextSegment*) – The text segment to add.

vertical_anchor

Sets the vertical anchor of the text layout.

width

Returns the width of the text.

> **Warning:** This is a slightly expensive operation and is not a simple getter.

> **Note:** If you want both width and height, use *dimensions*.

Type
int

wrap

Sets the wrapping style of the text layout. Make sure to also set the wrapping width using *width* for wrapping to work.

This must be set before adding any text segments!

2.1.6 Enums

class ril.**DisposalMethod**

The method used to dispose a frame before transitioning to the next frame in an image sequence.

Keep

Do not dispose the current frame. Usually not desired for transparent images.

Background

Dispose the current frame completely and replace it with the image's background color.

Previous

Dispose and replace the current frame with the previous frame.

class ril.**ResizeAlgorithm**

A filtering algorithm that is used to resize an image.

Nearest

A simple nearest neighbor algorithm. Although the fastest, this gives the lowest quality resizings.

When upscaling this is good if you want a "pixelated" effect with no aliasing.

Box

A box filter algorithm. Equivalent to the *Nearest* filter if you are upscaling.

Bilinear

A bilinear filter. Calculates output pixel value using linear interpolation on all pixels.

Hamming

While having similar performance as the *Bilinear* filter, this produces a sharper and usually considered better quality image than the *Bilinear* filter, but only when downscaling. This may give worse results than bilinear when upscaling.

Bicubic

A Catmull-Rom bicubic filter, which is the most common bicubic filtering algorithm. Just like all cubic filters, it uses cubic interpolation on all pixels to calculate output pixels.

Mitchell

A Mitchell-Netravali bicubic filter. Just like all cubic filters, it uses cubic interpolation on all pixels to calculate output pixels.

Lanczos3

A Lanczos filter with a window of 3. Calculates output pixel value using a high-quality Lanczos filter on all pixels.

class ril.**WrapStyle**

The wrapping style of text.

NoWrap

Do not wrap text.

Word

Wrap text on word boundaries.

Character

Wrap text on character boundaries.

class ril.**OverlayMode**

The mode to use when overlaying an image onto another image.

Overwrite

Overwrite the pixels of the image with the pixels of the overlay image.

Blend

Blend the pixels of the image with the pixels of the overlay image.

class ril.**HorizontalAnchor**

The horizontal anchor of text.

Left

Anchor text to the left.

Center

Anchor text to the center.

Right

Anchor text to the right.

class ril.**VerticalAnchor**

The vertical anchor of text.

Top

Anchor text to the top.

Center

Anchor text to the center.

Bottom

Anchor text to the bottom.

CHAPTER

THREE

INDEX

- genindex

INDEX