cairocffi

Release 0.8.0

Aug 17, 2017

Contents

cairocffi is a CFFI-based drop-in replacement for Pycairo, a set of Python bindings and object-oriented API for cairo. Cairo is a 2D vector graphics library with support for multiple backends including image buffers, PNG, PostScript, PDF, and SVG file output.

Additionally, the *cairocffi.pixbuf* module uses GDK-PixBuf to decode various image formats for use in cairo.

- Latest documentation
- Source code and issue tracker on GitHub.
- Install with `pip install cairocffi`
- Python 2.6, 2.7 and 3.2+. Tested with CPython and Pypy.
- API compatible with Pycairo.
- Works with any version of cairo.

CHAPTER 1

Documentation

Overview

Installing CFFI

cairocffi requires CFFI, which can be installed with pip but has its own dependencies that can be tricky to install.

- On Linux, install `python-dev` and `libffi-dev` from your system's package manager.
- On OS X, install `pkg-config` and `libffi`, for example with Homebrew. You may need to set the PKG_CONFIG_PATH environment variable.
- On Windows, consider using Christoph Gohlke's unofficial binary builds.

See CFFI's own documentation for details.

Installing cairocffi

Install with pip:

```
pip install cairocffi
```

This will automatically install CFFI.

cairocffi can also be setup to utizile XCB support via xcffib. This can also be installed automatically with pip:

```
pip install cairocffi[xcb]
```

In addition to other dependencies, this will install xcffib.

Importing

The module to import is named `cairocffi` in order to co-exist peacefully with Pycairo which uses `cairo`, but `cairo` is shorter and nicer to use:

```
import cairocffi as cairo
```

cairocffi will dynamically load cairo as a shared library at this point. If it fails to find it, you will see an exception like this:

```
OSError: library not found: 'cairo'
```

Make sure cairo is correctly installed and available through your system's usual mechanisms. On Linux, the `LD_LIBRARY_PATH` environment variable can be used to indicate where to find shared libraries.

Installing cairo on Windows

cairocffi needs a `libcairo-2.dll` file in a directory that is listed in the `PATH` environment variable.

Alexander Shaduri's GTK+ installer works. (Make sure to leave the *Set up PATH environment variable* checkbox checked.) Pycairo on Windows is sometimes compiled statically against cairo and may not provide a `.dll` file that cairocffi can use.

cairo versions

Cairo, pycairo, and cairocffi each have version numbers. The same cairocffi version can be used with a variety of cairo versions. For example, the *`Surface.set_mime_data()`* method is based on the `cairo_surface_set_mime_data()` C function, which is only available since cairo 1.10. You will get a run-time exception if you try to use it with an older cairo. You can however still use the rest of the API. There is no need for cairocffi's versions to be tied to cairo's versions.

Use *`cairo_version()`* to test the version number for cairo:

```
if cairocffi.cairo_version() > 11000:
    surface.set_mime_data('image/jpeg', jpeg_bytes)
```

Here are all the version numbers:

```
>>> print("The cairo version is %s, meaning %s."
...       % (cairocffi.cairo_version(), cairocffi.cairo_version_string())
The cairo version is 11402, meaning 1.14.02.
>>> print("The latest pycairo version this cairocffi version is compatible with is %s.
↪"
...       % cairo.version)
The latest pycairo version this cairocffi version is compatible with is 1.10.0.
>>> print("The cairocffi version is %s." % cairo.VERSION)
The cairocffi version is 0.7.2
```

cairocffi is tested with both cairo 1.8.2 and the latest (1.12.8 as of this writing.)

Compatibility with Pycairo

cairocffi's Python API is compatible with Pycairo. Please file a bug if you find incompatibilities.

In your own code that uses Pycairo, you should be able to just change the imports from `import cairo` to `import cairocffi as cairo` as above. If it's not your own code that imports Pycairo, the *`install_as_pycairo()`* function can help:

```
import cairocffi
cairocffi.install_as_pycairo()
import cairo
assert cairo is cairocffi
```

Alternatively, add a `cairo.py` file somewhere in your `sys.path`, so that it's imported instead of pycairo:

```
from cairocffi import *
```

It is also possible to *convert pycairo contexts to cairocffi*.

Basic usage

For doing something useful with cairo, you need at least a surface and a context:

```
import cairocffi as cairo

surface = cairo.ImageSurface(cairo.FORMAT_ARGB32, 300, 200)
context = cairo.Context(surface)
with context:
    context.set_source_rgb(1, 1, 1)  # White
    context.paint()
# Restore the default source which is black.
context.move_to(90, 140)
context.rotate(-0.5)
context.set_font_size(20)
context.show_text(u'Hi from cairo!')
surface.write_to_png('example.png')
```

The *Surface* represents the target. There are various types of surface for various output backends. The *Context* holds internal state and is use for drawing. We're only using solid colors here, but more complex *Pattern* types are also available.

All the details are in *Python API reference*.

Python API reference

Meta

cairocffi.**install_as_pycairo**()
: Install cairocffi so that `import cairo` imports it.

  cairoffi's API is compatible with pycairo as much as possible.

cairocffi.**cairo_version**()
: Return the cairo version number as a single integer, such as 11208 for `1.12.8`. Major, minor and micro versions are "worth" 10000, 100 and 1 respectively.

  Can be useful as a guard for method not available in older cairo versions:

  ```
  if cairo_version() >= 11000:
      surface.set_mime_data('image/jpeg', jpeg_bytes)
  ```

cairocffi.**cairo_version_string**()
: Return the cairo version number as a string, such as `1.12.8`.

exception cairocffi.**CairoError**(*message*, *status*)
Raised when cairo returns an error status.

cairocffi.**Error**
An alias for *CairoError*, for compatibility with pycairo.

Surfaces

class cairocffi.**Surface**
The base class for all surface types.

Should not be instantiated directly, but see *CFFI API*. An instance may be returned for cairo surface types that are not (yet) defined in cairocffi.

A *Surface* represents an image, either as the destination of a drawing operation or as source when drawing onto another surface. To draw to a *Surface*, create a cairo *Context* with the surface as the target.

There are different sub-classes of *Surface* for different drawing backends; for example, *ImageSurface* is a bitmap image in memory.

The initial contents of a surface after creation depend upon the manner of its creation. If cairo creates the surface and backing storage for the user, it will be initially cleared; for example, *ImageSurface* and *create_similar()*. Alternatively, if the user passes in a reference to some backing storage and asks cairo to wrap that in a *Surface*, then the contents are not modified; for example, *ImageSurface* with a `data` argument.

create_similar(*content*, *width*, *height*)
Create a new surface that is as compatible as possible for uploading to and the use in conjunction with this surface. For example the new surface will have the same fallback resolution and *FontOptions*. Generally, the new surface will also use the same backend as other, unless that is not possible for some reason.

Initially the surface contents are all 0 (transparent if contents have transparency, black otherwise.)

Use *create_similar_image()* if you need an image surface which can be painted quickly to the target surface.

Parameters

- **content** (*str*) – the *Content* string for the new surface.
- **width** (*int*) – width of the new surface (in device-space units)
- **height** (*int*) – height of the new surface (in device-space units)

Returns A new instance of *Surface* or one of its subclasses.

create_similar_image(*content*, *width*, *height*)
Create a new image surface that is as compatible as possible for uploading to and the use in conjunction with this surface. However, this surface can still be used like any normal image surface.

Initially the surface contents are all 0 (transparent if contents have transparency, black otherwise.)

Use *create_similar()* if you don't need an image surface.

Parameters

- **format** (*str*) – the *Pixel format* string for the new surface
- **width** (*int*) – width of the new surface, (in device-space units)
- **height** (*int*) – height of the new surface (in device-space units)

Returns A new *ImageSurface* instance.

create_for_rectangle(*x*, *y*, *width*, *height*)
Create a new surface that is a rectangle within this surface. All operations drawn to this surface are then clipped and translated onto the target surface. Nothing drawn via this sub-surface outside of its bounds is drawn onto the target surface, making this a useful method for passing constrained child surfaces to library routines that draw directly onto the parent surface, i.e. with no further backend allocations, double buffering or copies.

Note: As of cairo 1.12, the semantics of subsurfaces have not been finalized yet unless the rectangle is in full device units, is contained within the extents of the target surface, and the target or subsurface's device transforms are not changed.

Parameters

- **x** (*float*) – The x-origin of the sub-surface from the top-left of the target surface (in device-space units)
- **y** (*float*) – The y-origin of the sub-surface from the top-left of the target surface (in device-space units)
- **width** (*float*) – Width of the sub-surface (in device-space units)
- **height** (*float*) – Height of the sub-surface (in device-space units)

Returns A new *Surface* object.

New in cairo 1.10.

get_content()
Returns the *Content* string of this surface, which indicates whether the surface contains color and/or alpha information.

has_show_text_glyphs()
Returns whether the surface supports sophisticated *Context.show_text_glyphs()* operations. That is, whether it actually uses the text and cluster data provided to a *Context.show_text_glyphs()* call.

Note: Even if this method returns `False`, *Context.show_text_glyphs()* operation targeted at surface will still succeed. It just will act like a *Context.show_glyphs()* operation. Users can use this method to avoid computing UTF-8 text and cluster mapping if the target surface does not use it.

---

set_device_offset(*x_offset*, *y_offset*)
Sets an offset that is added to the device coordinates determined by the CTM when drawing to surface. One use case for this method is when we want to create a *Surface* that redirects drawing for a portion of an onscreen surface to an offscreen surface in a way that is completely invisible to the user of the cairo API. Setting a transformation via *Context.translate()* isn't sufficient to do this, since methods like *Context.device_to_user()* will expose the hidden offset.

Note that the offset affects drawing to the surface as well as using the surface in a source pattern.

Parameters

- **x_offset** – The offset in the X direction, in device units
- **y_offset** – The offset in the Y direction, in device units

get_device_offset()
Returns the previous device offset set by *set_device_offset()*.

Returns `(x_offset, y_offset)`

set_fallback_resolution(*x_pixels_per_inch*, *y_pixels_per_inch*)
Set the horizontal and vertical resolution for image fallbacks.

When certain operations aren't supported natively by a backend, cairo will fallback by rendering operations to an image and then overlaying that image onto the output. For backends that are natively vector-oriented, this method can be used to set the resolution used for these image fallbacks, (larger values will result in more detailed images, but also larger file sizes).

Some examples of natively vector-oriented backends are the ps, pdf, and svg backends.

For backends that are natively raster-oriented, image fallbacks are still possible, but they are always performed at the native device resolution. So this method has no effect on those backends.

Note: The fallback resolution only takes effect at the time of completing a page (with *show_page()* or *copy_page()*) so there is currently no way to have more than one fallback resolution in effect on a single page.

The default fallback resoultion is 300 pixels per inch in both dimensions.

Parameters

- **x_pixels_per_inch** (*float*) – horizontal resolution in pixels per inch
- **y_pixels_per_inch** (*float*) – vertical resolution in pixels per inch

get_fallback_resolution()
Returns the previous fallback resolution set by *set_fallback_resolution()*, or default fallback resolution if never set.

Returns `(x_pixels_per_inch, y_pixels_per_inch)`

get_font_options()
Retrieves the default font rendering options for the surface.

This allows display surfaces to report the correct subpixel order for rendering on them, print surfaces to disable hinting of metrics and so forth. The result can then be used with *ScaledFont*.

Returns A new *FontOptions* object.

set_mime_data(*mime_type*, *data*)
Attach an image in the format `mime_type` to this surface.

To remove the data from a surface, call this method with same mime type and `None` for data.

The attached image (or filename) data can later be used by backends which support it (currently: PDF, PS, SVG and Win32 Printing surfaces) to emit this data instead of making a snapshot of the surface. This approach tends to be faster and requires less memory and disk space.

The recognized MIME types are the following:

"image/png" The Portable Network Graphics image file format (ISO/IEC 15948).

"image/jpeg" The Joint Photographic Experts Group (JPEG) image coding standard (ISO/IEC 10918-1).

"image/jp2" The Joint Photographic Experts Group (JPEG) 2000 image coding standard (ISO/IEC 15444-1).

"text/x-uri" URL for an image file (unofficial MIME type).

See corresponding backend surface docs for details about which MIME types it can handle. Caution: the associated MIME data will be discarded if you draw on the surface afterwards. Use this method with care.

Parameters

- **mime_type** (*ASCII string*) – The MIME type of the image data.
- **data** (*bytes*) – The image data to attach to the surface.

New in cairo 1.10.

get_mime_data(*mime_type*)
Return mime data previously attached to surface using the specified mime type.

Parameters **mime_type** (*ASCII string*) – The MIME type of the image data.

Returns A CFFI buffer object, or `None` if no data has been attached with the given mime type.

New in cairo 1.10.

supports_mime_type(*mime_type*)
Return whether surface supports `mime_type`.

Parameters **mime_type** (*ASCII string*) – The MIME type of the image data.

New in cairo 1.12.

mark_dirty()
Tells cairo that drawing has been done to surface using means other than cairo, and that cairo should reread any cached areas. Note that you must call *flush()* before doing such drawing.

mark_dirty_rectangle(*x*, *y*, *width*, *height*)
Like *mark_dirty()*, but drawing has been done only to the specified rectangle, so that cairo can retain cached contents for other parts of the surface.

Any cached clip set on the surface will be reset by this method, to make sure that future cairo calls have the clip set that they expect.

Parameters

- **x** (*float*) – X coordinate of dirty rectangle.
- **y** (*float*) – Y coordinate of dirty rectangle.
- **width** (*float*) – Width of dirty rectangle.
- **height** (*float*) – Height of dirty rectangle.

show_page()
Emits and clears the current page for backends that support multiple pages. Use *copy_page()* if you don't want to clear the page.

Context.show_page() is a convenience method for this.

copy_page()
Emits the current page for backends that support multiple pages, but doesn't clear it, so that the contents of the current page will be retained for the next page.

Use *show_page()* if you want to get an empty page after the emission.

flush()
Do any pending drawing for the surface and also restore any temporary modifications cairo has made to the surface's state. This method must be called before switching from drawing on the surface with cairo to drawing on it directly with native APIs. If the surface doesn't support direct access, then this method does nothing.

finish()
This method finishes the surface and drops all references to external resources. For example, for the Xlib backend it means that cairo will no longer access the drawable, which can be freed. After calling *finish()* the only valid operations on a surface are getting and setting user data, flushing and finishing it. Further drawing to the surface will not affect the surface but will instead trigger a *CairoError* with a SURFACE_FINISHED status.

When the surface is garbage-collected, cairo will call *finish()* if it hasn't been called already, before freeing the resources associated with the surface.

write_to_png(*target=None*)
Writes the contents of surface as a PNG image.

Parameters **target** – A filename, a binary mode file-like object with a `write()` method, or `None`.

Returns If `target` is `None`, return the PNG contents as a byte string.

ImageSurface

class cairocffi.**ImageSurface**(*format*, *width*, *height*, *data=None*, *stride=None*)
Creates an image surface of the specified format and dimensions.

If `data` is not `None` its initial contents will be used as the initial image contents; you must explicitly clear the buffer, using, for example, *Context.rectangle()* and *Context.fill()* if you want it cleared.

Note: Currently only `array.array` buffers are supported on PyPy.

Otherwise, the surface contents are all initially 0. (Specifically, within each pixel, each color or alpha channel belonging to format will be 0. The contents of bits within a pixel, but not belonging to the given format are undefined).

Parameters

- **format** (*str*) – *Pixel format* string for the surface to create.
- **width** (*int*) – Width of the surface, in pixels.
- **height** (*int*) – Height of the surface, in pixels.
- **data** – Buffer supplied in which to write contents, or `None` to create a new buffer.
- **stride** (*int*) – The number of bytes between the start of rows in the buffer as allocated. This value should always be computed by *format_stride_for_width()* before allocating the data buffer. If omitted but `data` is given, *format_stride_for_width()* is used.

classmethod create_for_data(*data*, *format*, *width*, *height*, *stride=None*)
Same as `ImageSurface(format, width, height, data, stride)`. Exists for compatibility with pycairo.

static format_stride_for_width(*format*, *width*)
This method provides a stride value (byte offset between rows) that will respect all alignment requirements of the accelerated image-rendering code within cairo. Typical usage will be of the form:

```
from cairocffi import ImageSurface
stride = ImageSurface.format_stride_for_width(format, width)
data = bytearray(stride * height)
surface = ImageSurface(format, width, height, data, stride)
```

Parameters

- **format** (*str*) – A *Pixel format* string.
- **width** (*int*) – The desired width of the surface, in pixels.

Returns The appropriate stride to use given the desired format and width, or -1 if either the format is invalid or the width too large.

classmethod create_from_png(*source*)
Decode a PNG file into a new image surface.

Parameters **source** – A filename or a binary mode file-like object with a `read()` method. If you already have a byte string in memory, use `io.BytesIO`.

Returns A new *ImageSurface* instance.

get_data()
Return the buffer pointing to the image's pixel data, encoded according to the surface's *Pixel format* string.

A call to `flush()` is required before accessing the pixel data to ensure that all pending drawing operations are finished. A call to *mark_dirty()* is required after the data is modified.

Returns A read-write CFFI buffer object.

get_format()
Return the *Pixel format* string of the surface.

get_width()
Return the width of the surface, in pixels.

get_height()
Return the width of the surface, in pixels.

get_stride()
Return the stride of the image surface in bytes (or 0 if surface is not an image surface).

The stride is the distance in bytes from the beginning of one row of the image data to the beginning of the next row.

PDFSurface

class cairocffi.**PDFSurface**(*target*, *width_in_points*, *height_in_points*)
Creates a PDF surface of the specified size in PostScript points to be written to `target`.

Note that the size of individual pages of the PDF output can vary. See *set_size()*.

The PDF surface backend recognizes the following MIME types for the data attached to a surface (see *set_mime_data()*) when it is used as a source pattern for drawing on this surface: `image/jpeg` and `image/jp2`. If any of them is specified, the PDF backend emits an image with the content of MIME data (with the `/DCTDecode` or `/JPXDecode` filter, respectively) instead of a surface snapshot (with the `/FlateDecode` filter), which typically produces PDF with a smaller file size.

`target` can be `None` to specify no output. This will generate a surface that may be queried and used as a source, without generating a temporary file.

Parameters

- **target** – A filename, a binary mode file-like object with a `write()` method, or `None`.
- **width_in_points** (*float*) – Width of the surface, in points (1 point == 1/72.0 inch)
- **height_in_points** (*float*) – Height of the surface, in points (1 point == 1/72.0 inch)

set_size(*width_in_points*, *height_in_points*)
Changes the size of a PDF surface for the current (and subsequent) pages.

This method should only be called before any drawing operations have been performed on the current page. The simplest way to do this is to call this method immediately after creating the surface or immediately after completing a page with either *show_page()* or *copy_page()*.

Parameters

- **width_in_points** (*float*) – New width of the page, in points (1 point == 1/72.0 inch)
- **height_in_points** (*float*) – New height of the page, in points (1 point == 1/72.0 inch)

restrict_to_version(*version*)
Restricts the generated PDF file to `version`.

See *get_versions()* for a list of available version values that can be used here.

This method should only be called before any drawing operations have been performed on the given surface. The simplest way to do this is to call this method immediately after creating the surface.

Parameters **version** – A *PDF version* string.

New in cairo 1.10.

static get_versions()
Return the list of supported PDF versions. See *restrict_to_version()*.

Returns A list of *PDF version* strings.

New in cairo 1.10.

static version_to_string(*version*)
Return the string representation of the given *PDF version*. See *get_versions()* for a way to get the list of valid version ids.

New in cairo 1.10.

PSSurface

class cairocffi.**PSSurface**(*target*, *width_in_points*, *height_in_points*)
Creates a PostScript surface of the specified size in PostScript points to be written to `target`.

Note that the size of individual pages of the PostScript output can vary. See *set_size()*.

`target` can be `None` to specify no output. This will generate a surface that may be queried and used as a source, without generating a temporary file.

The PostScript surface backend recognizes the `image/jpeg` MIME type for the data attached to a surface (see *set_mime_data()*) when it is used as a source pattern for drawing on this surface. If it is specified, the PostScript backend emits an image with the content of MIME data (with the `/DCTDecode` filter) instead of a surface snapshot (with the `/FlateDecode` filter), which typically produces PostScript with a smaller file size.

Parameters

- **target** – A filename, a binary mode file-like object with a `write()` method, or `None`.
- **width_in_points** (*float*) – Width of the surface, in points (1 point == 1/72.0 inch)
- **height_in_points** (*float*) – Height of the surface, in points (1 point == 1/72.0 inch)

dsc_comment(*comment*)
Emit a comment into the PostScript output for the given surface.

The comment is expected to conform to the PostScript Language Document Structuring Conventions (DSC). Please see that manual for details on the available comments and their meanings. In particular, the `%%IncludeFeature` comment allows a device-independent means of controlling printer device features. So the PostScript Printer Description Files Specification will also be a useful reference.

The comment string must begin with a percent character (%) and the total length of the string (including any initial percent characters) must not exceed 255 bytes. Violating either of these conditions will place surface into an error state. But beyond these two conditions, this method will not enforce conformance of the comment with any particular specification.

The comment string should not have a trailing newline.

The DSC specifies different sections in which particular comments can appear. This method provides for comments to be emitted within three sections: the header, the Setup section, and the PageSetup section. Comments appearing in the first two sections apply to the entire document while comments in the BeginPageSetup section apply only to a single page.

For comments to appear in the header section, this method should be called after the surface is created, but before a call to *dsc_begin_setup()*.

For comments to appear in the Setup section, this method should be called after a call to *dsc_begin_setup()* but before a call to *dsc_begin_page_setup()*.

For comments to appear in the PageSetup section, this method should be called after a call to *dsc_begin_page_setup()*.

Note that it is only necessary to call *dsc_begin_page_setup()* for the first page of any surface. After a call to *show_page()* or *copy_page()* comments are unambiguously directed to the PageSetup section of the current page. But it doesn't hurt to call this method at the beginning of every page as that consistency may make the calling code simpler.

As a final note, cairo automatically generates several comments on its own. As such, applications must not manually generate any of the following comments:

Header section: `%!PS-Adobe-3.0`, `%%Creator`, `%%CreationDate`, `%%Pages`, `%%BoundingBox`, `%%DocumentData`, `%%LanguageLevel`, `%%EndComments`.

Setup section: `%%BeginSetup`, `%%EndSetup`.

PageSetup section: `%%BeginPageSetup`, `%%PageBoundingBox`, `%%EndPageSetup`.

Other sections: `%%BeginProlog`, `%%EndProlog`, `%%Page`, `%%Trailer`, `%%EOF`.

dsc_begin_setup()
Indicate that subsequent calls to *dsc_comment()* should direct comments to the Setup section of the PostScript output.

This method should be called at most once per surface, and must be called before any call to *dsc_begin_page_setup()* and before any drawing is performed to the surface.

See *dsc_comment()* for more details.

dsc_begin_page_setup()
Indicate that subsequent calls to *dsc_comment()* should direct comments to the PageSetup section of the PostScript output.

This method is only needed for the first page of a surface. It must be called after any call to *dsc_begin_setup()* and before any drawing is performed to the surface.

See *dsc_comment()* for more details.

set_eps(*eps*)
If `eps` is True, the PostScript surface will output Encapsulated PostScript.

This method should only be called before any drawing operations have been performed on the current page. The simplest way to do this is to call this method immediately after creating the surface. An Encapsulated PostScript file should never contain more than one page.

get_eps()
Check whether the PostScript surface will output Encapsulated PostScript.

set_size(*width_in_points*, *height_in_points*)
Changes the size of a PostScript surface for the current (and subsequent) pages.

This method should only be called before any drawing operations have been performed on the current page. The simplest way to do this is to call this method immediately after creating the surface or immediately after completing a page with either *show_page()* or *copy_page()*.

Parameters

- **width_in_points** (*float*) – New width of the page, in points (1 point == 1/72.0 inch)
- **height_in_points** (*float*) – New height of the page, in points (1 point == 1/72.0 inch)

restrict_to_level(*level*)
Restricts the generated PostScript file to `level`.

See *get_levels()* for a list of available level values that can be used here.

This method should only be called before any drawing operations have been performed on the given surface. The simplest way to do this is to call this method immediately after creating the surface.

Parameters **version** – A *PostScript level* string.

static get_levels()
Return the list of supported PostScript levels. See *restrict_to_level()*.

Returns A list of *PostScript level* strings.

static ps_level_to_string(*level*)
Return the string representation of the given *PostScript level*. See *get_levels()* for a way to get the list of valid level ids.

SVGSurface

class cairocffi.**SVGSurface**(*target*, *width_in_points*, *height_in_points*)
Creates a SVG surface of the specified size in points to be written to `target`.

`target` can be `None` to specify no output. This will generate a surface that may be queried and used as a source, without generating a temporary file.

The SVG surface backend recognizes the following MIME types for the data attached to a surface (see *set_mime_data()*) when it is used as a source pattern for drawing on this surface: `image/png`, `image/jpeg` and `text/x-uri`. If any of them is specified, the SVG backend emits a href with the content of MIME data instead of a surface snapshot (PNG, Base64-encoded) in the corresponding image tag.

The unofficial MIME type `text/x-uri` is examined first. If present, the URL is emitted as is: assuring the correctness of URL is left to the client code.

If `text/x-uri` is not present, but `image/jpeg` or `image/png` is specified, the corresponding data is Base64-encoded and emitted.

Parameters

- **target** – A filename, a binary mode file-like object with a `write()` method, or `None`.
- **width_in_points** (*float*) – Width of the surface, in points (1 point == 1/72.0 inch)
- **height_in_points** (*float*) – Height of the surface, in points (1 point == 1/72.0 inch)

restrict_to_version(*version*)
Restricts the generated SVG file to `version`.

See *get_versions()* for a list of available version values that can be used here.

This method should only be called before any drawing operations have been performed on the given surface. The simplest way to do this is to call this method immediately after creating the surface.

Parameters **version** – A *SVG version* string.

static get_versions()
Return the list of supported SVG versions. See *restrict_to_version()*.

Returns A list of *SVG version* strings.

static version_to_string(*version*)
Return the string representation of the given *SVG version*. See *get_versions()* for a way to get the list of valid version ids.

RecordingSurface

class cairocffi.**RecordingSurface**(*content*, *extents*)
A recording surface is a surface that records all drawing operations at the highest level of the surface backend interface, (that is, the level of paint, mask, stroke, fill, and show_text_glyphs). The recording surface can then be "replayed" against any target surface by using it as a source surface.

If you want to replay a surface so that the results in `target` will be identical to the results that would have been obtained if the original operations applied to the recording surface had instead been applied to the target surface, you can use code like this:

```
context = Context(target)
context.set_source_surface(recording_surface, 0, 0)
context.paint()
```

A recording surface is logically unbounded, i.e. it has no implicit constraint on the size of the drawing surface. However, in practice this is rarely useful as you wish to replay against a particular target surface with known bounds. For this case, it is more efficient to specify the target extents to the recording surface upon creation.

The recording phase of the recording surface is careful to snapshot all necessary objects (paths, patterns, etc.), in order to achieve accurate replay.

Parameters

- **content** – The *Content* string of the recording surface
- **extents** – The extents to record as a `(x, y, width, height)` tuple of floats in device units, or `None` to record unbounded operations. `(x, y)` are the coordinates of the top-left corner of the rectangle, `(width, height)` its dimensions.

New in cairo 1.10

New in cairocffi 0.2

get_extents()
Return the extents of the recording-surface.

Returns A `(x, y, width, height)` tuple of floats, or `None` if the surface is unbounded.

New in cairo 1.12

ink_extents()
Measures the extents of the operations stored within the recording-surface. This is useful to compute the required size of an image surface (or equivalent) into which to replay the full sequence of drawing operations.

Returns A `(x, y, width, height)` tuple of floats.

Win32PrintingSurface

class cairocffi.**Win32PrintingSurface**(*hdc*)
Creates a cairo surface that targets the given DC.

The DC will be queried for its initial clip extents, and this will be used as the size of the cairo surface. The DC should be a printing DC; antialiasing will be ignored, and GDI will be used as much as possible to draw to the surface.

The returned surface will be wrapped using the paginated surface to provide correct complex rendering behaviour; cairo_surface_show_page() and associated methods must be used for correct output.

Parameters **hdc** (*int*) – The DC to create a surface for, as obtained from `win32gui.CreateDC()`. **Note**: this unsafely inteprets an integer as a pointer. Make sure it actually points to a valid DC!

New in cairocffi 0.6

Context

class cairocffi.**Context**(*target*)
A *Context* contains the current state of the rendering device, including coordinates of yet to be drawn shapes.

Cairo contexts are central to cairo and all drawing with cairo is always done to a *Context* object.

Parameters **target** – The target *Surface* object.

Cairo contexts can be used as Python context managers. See *save()*.

get_target()
Return this context's target surface.

Returns An instance of *Surface* or one of its sub-classes, a new Python object referencing the existing cairo surface.

save()
Makes a copy of the current state of this context and saves it on an internal stack of saved states. When *restore()* is called, the context will be restored to the saved state. Multiple calls to *save()* and *restore()* can be nested; each call to *restore()* restores the state from the matching paired *save()*.

Instead of using *save()* and *restore()* directly, it is recommended to use a with statement:

```
with context:
    do_something(context)
```

... which is equivalent to:

```
context.save()
try:
    do_something(context)
finally:
    context.restore()
```

restore()
Restores the context to the state saved by a preceding call to *save()* and removes that state from the stack of saved states.

push_group()
Temporarily redirects drawing to an intermediate surface known as a group. The redirection lasts until the group is completed by a call to *pop_group()* or *pop_group_to_source()*. These calls provide the result of any drawing to the group as a pattern, (either as an explicit object, or set as the source pattern).

This group functionality can be convenient for performing intermediate compositing. One common use of a group is to render objects as opaque within the group, (so that they occlude each other), and then blend the result with translucence onto the destination.

Groups can be nested arbitrarily deep by making balanced calls to *push_group()* / *pop_group()*. Each call pushes / pops the new target group onto / from a stack.

The `group()` method calls *save()* so that any changes to the graphics state will not be visible outside the group, (the pop_group methods call *restore()*).

By default the intermediate group will have a content type of *COLOR_ALPHA*. Other content types can be chosen for the group by using *push_group_with_content()* instead.

As an example, here is how one might fill and stroke a path with translucence, but without any portion of the fill being visible under the stroke:

```
context.push_group()
context.set_source(fill_pattern)
context.fill_preserve()
context.set_source(stroke_pattern)
context.stroke()
context.pop_group_to_source()
context.paint_with_alpha(alpha)
```

push_group_with_content(*content*)
Temporarily redirects drawing to an intermediate surface known as a group. The redirection lasts until the group is completed by a call to *pop_group()* or *pop_group_to_source()*. These calls provide the result of any drawing to the group as a pattern, (either as an explicit object, or set as the source pattern).

The group will have a content type of `content`. The ability to control this content type is the only distinction between this method and *push_group()* which you should see for a more detailed description of group rendering.

Parameters **content** – A *Content* string.

pop_group()
Terminates the redirection begun by a call to *push_group()* or *push_group_with_content()* and returns a new pattern containing the results of all drawing operations performed to the group.

The *pop_group()* method calls *restore()*, (balancing a call to *save()* by the push_group method), so that any changes to the graphics state will not be visible outside the group.

Returns A newly created *SurfacePattern* containing the results of all drawing operations performed to the group.

pop_group_to_source()
Terminates the redirection begun by a call to *push_group()* or *push_group_with_content()* and installs the resulting pattern as the source pattern in the given cairo context.

The behavior of this method is equivalent to:

```
context.set_source(context.pop_group())
```

get_group_target()
Returns the current destination surface for the context. This is either the original target surface as passed to *Context* or the target surface for the current group as started by the most recent call to *push_group()* or *push_group_with_content()*.

set_source_rgba(*red*, *green*, *blue*, *alpha=1*)
Sets the source pattern within this context to a solid color. This color will then be used for any subsequent drawing operation until a new source pattern is set.

The color and alpha components are floating point numbers in the range 0 to 1. If the values passed in are outside that range, they will be clamped.

The default source pattern is opaque black, (that is, it is equivalent to `context.set_source_rgba(0, 0, 0)`).

Parameters

- **red** (*float*) – Red component of the color.
- **green** (*float*) – Green component of the color.
- **blue** (*float*) – Blue component of the color.
- **alpha** (*float*) – Alpha component of the color. 1 (the default) is opaque, 0 fully transparent.

set_source_rgb(*red*, *green*, *blue*)
Same as *set_source_rgba()* with alpha always 1. Exists for compatibility with pycairo.

set_source_surface(*surface*, *x=0*, *y=0*)
This is a convenience method for creating a pattern from surface and setting it as the source in this context with *set_source()*.

The `x` and `y` parameters give the user-space coordinate at which the surface origin should appear. (The surface origin is its upper-left corner before any transformation has been applied.) The `x` and `y` parameters are negated and then set as translation values in the pattern matrix.

Other than the initial translation pattern matrix, as described above, all other pattern attributes, (such as its extend mode), are set to the default values as in *SurfacePattern*. The resulting pattern can be queried with *get_source()* so that these attributes can be modified if desired, (eg. to create a repeating pattern with *Pattern.set_extend()*).

Parameters

- **surface** – A *Surface* to be used to set the source pattern.
- **x** (*float*) – User-space X coordinate for surface origin.
- **y** (*float*) – User-space Y coordinate for surface origin.

set_source(*source*)
Sets the source pattern within this context to `source`. This pattern will then be used for any subsequent drawing operation until a new source pattern is set.

Note: The pattern's transformation matrix will be locked to the user space in effect at the time of *set_source()*. This means that further modifications of the current transformation matrix will not affect the source pattern. See *Pattern.set_matrix()*.

The default source pattern is opaque black, (that is, it is equivalent to `context.set_source_rgba(0, 0, 0)`).

Parameters **source** – A *Pattern* to be used as the source for subsequent drawing operations.

get_source()
Return this context's source.

Returns An instance of *Pattern* or one of its sub-classes, a new Python object referencing the existing cairo pattern.

set_antialias(*antialias*)
Set the *Antialiasing mode* of the rasterizer used for drawing shapes. This value is a hint, and a particular backend may or may not support a particular value. At the current time, no backend supports *SUBPIXEL* when drawing shapes.

Note that this option does not affect text rendering, instead see *FontOptions.set_antialias()*.

Parameters **antialias** – An *Antialiasing mode* string.

get_antialias()
Return the *Antialiasing mode* string.

set_dash(*dashes*, *offset=0*)
Sets the dash pattern to be used by *stroke()*. A dash pattern is specified by dashes, a list of positive values. Each value provides the length of alternate "on" and "off" portions of the stroke. `offset` specifies an offset into the pattern at which the stroke begins.

Each "on" segment will have caps applied as if the segment were a separate sub-path. In particular, it is valid to use an "on" length of 0 with *LINE_CAP_ROUND* or *LINE_CAP_SQUARE* in order to distributed dots or squares along a path.

Note: The length values are in user-space units as evaluated at the time of stroking. This is not necessarily the same as the user space at the time of *set_dash()*.

If `dashes` is empty dashing is disabled. If it is of length 1 a symmetric pattern is assumed with alternating on and off portions of the size specified by the single value.

Parameters

- **dashes** – A list of floats specifying alternate lengths of on and off stroke portions.
- **offset** (*float*) – An offset into the dash pattern at which the stroke should start.

Raises *CairoError* if any value in dashes is negative, or if all values are 0. The context will be put into an error state.

get_dash()
Return the current dash pattern.

Returns A `(dashes, offset)` tuple of a list and a float. `dashes` is a list of floats, empty if no dashing is in effect.

get_dash_count()
Same as `len(context.get_dash()[0])`.

set_fill_rule(*fill_rule*)
Set the current *Fill rule* within the cairo context. The fill rule is used to determine which regions are inside

or outside a complex (potentially self-intersecting) path. The current fill rule affects both *fill()* and *clip()*.

The default fill rule is *WINDING*.

Parameters **fill_rule** – A *Fill rule* string.

get_fill_rule()
Return the current *Fill rule* string.

set_line_cap(*line_cap*)
Set the current *Line cap style* within the cairo context. As with the other stroke parameters, the current line cap style is examined by *stroke()*, *stroke_extents()*, and stroke_to_path(), but does not have any effect during path construction.

The default line cap is *BUTT*.

Parameters **line_cap** – A *Line cap style* string.

get_line_cap()
Return the current *Line cap style* string.

set_line_join(*line_join*)
Set the current *Line join style* within the cairo context. As with the other stroke parameters, the current line cap style is examined by *stroke()*, *stroke_extents()*, and stroke_to_path(), but does not have any effect during path construction.

The default line cap is *MITER*.

Parameters **line_join** – A *Line join style* string.

get_line_join()
Return the current *Line join style* string.

set_line_width(*width*)
Sets the current line width within the cairo context. The line width value specifies the diameter of a pen that is circular in user space, (though device-space pen may be an ellipse in general due to scaling / shear / rotation of the CTM).

Note: When the description above refers to user space and CTM it refers to the user space and CTM in effect at the time of the stroking operation, not the user space and CTM in effect at the time of the call to *set_line_width()*. The simplest usage makes both of these spaces identical. That is, if there is no change to the CTM between a call to *set_line_width()* and the stroking operation, then one can just pass user-space values to *set_line_width()* and ignore this note.

As with the other stroke parameters, the current line cap style is examined by *stroke()*, *stroke_extents()*, and stroke_to_path(), but does not have any effect during path construction.

The default line width value is 2.0.

Parameters **width** (*float*) – The new line width.

get_line_width()
Return the current line width as a float.

set_miter_limit(*limit*)
Sets the current miter limit within the cairo context.

If the current line join style is set to *MITER* (see *set_line_join()*), the miter limit is used to determine whether the lines should be joined with a bevel instead of a miter. Cairo divides the length of the miter by the line width. If the result is greater than the miter limit, the style is converted to a bevel.

As with the other stroke parameters, the current line cap style is examined by *stroke()*, *stroke_extents()*, and `stroke_to_path()`, but does not have any effect during path construction.

The default miter limit value is 10.0, which will convert joins with interior angles less than 11 degrees to bevels instead of miters. For reference, a miter limit of 2.0 makes the miter cutoff at 60 degrees, and a miter limit of 1.414 makes the cutoff at 90 degrees.

A miter limit for a desired angle can be computed as: `miter_limit = 1. / sin(angle / 2.)`

Parameters **limit** (*float*) – The miter limit to set.

get_miter_limit()
Return the current miter limit as a float.

set_operator(*operator*)
Set the current *Compositiong operator* to be used for all drawing operations.

The default operator is *OVER*.

Parameters **operator** – A *Compositiong operator* string.

get_operator()
Return the current *Compositiong operator* string.

set_tolerance(*tolerance*)
Sets the tolerance used when converting paths into trapezoids. Curved segments of the path will be subdivided until the maximum deviation between the original path and the polygonal approximation is less than tolerance. The default value is 0.1. A larger value will give better performance, a smaller value, better appearance. (Reducing the value from the default value of 0.1 is unlikely to improve appearance significantly.) The accuracy of paths within Cairo is limited by the precision of its internal arithmetic, and the prescribed tolerance is restricted to the smallest representable internal value.

Parameters **tolerance** (*float*) – The tolerance, in device units (typically pixels)

get_tolerance()
Return the current tolerance as a float.

translate(*tx*, *ty*)
Modifies the current transformation matrix (CTM) by translating the user-space origin by `(tx, ty)`. This offset is interpreted as a user-space coordinate according to the CTM in place before the new call to *translate()*. In other words, the translation of the user-space origin takes place after any existing transformation.

Parameters

- **tx** (*float*) – Amount to translate in the X direction
- **ty** (*float*) – Amount to translate in the Y direction

scale(*sx*, *sy=None*)
Modifies the current transformation matrix (CTM) by scaling the X and Y user-space axes by `sx` and `sy` respectively. The scaling of the axes takes place after any existing transformation of user space.

If `sy` is omitted, it is the same as `sx` so that scaling preserves aspect ratios.

Parameters

- **sx** (*float*) – Scale factor in the X direction.

- **sy** (*float*) – Scale factor in the Y direction.

rotate(*radians*)
Modifies the current transformation matrix (CTM) by rotating the user-space axes by angle `radians`. The rotation of the axes takes places after any existing transformation of user space.

Parameters **radians** (*float*) – Angle of rotation, in radians. The direction of rotation is defined such that positive angles rotate in the direction from the positive X axis toward the positive Y axis. With the default axis orientation of cairo, positive angles rotate in a clockwise direction.

transform(*matrix*)
Modifies the current transformation matrix (CTM) by applying `matrix` as an additional transformation. The new transformation of user space takes place after any existing transformation.

Parameters **matrix** – A transformation *Matrix* to be applied to the user-space axes.

set_matrix(*matrix*)
Modifies the current transformation matrix (CTM) by setting it equal to `matrix`.

Parameters **matrix** – A transformation *Matrix* from user space to device space.

get_matrix()
Return a copy of the current transformation matrix (CTM).

identity_matrix()
Resets the current transformation matrix (CTM) by setting it equal to the identity matrix. That is, the user-space and device-space axes will be aligned and one user-space unit will transform to one device-space unit.

user_to_device(*x*, *y*)
Transform a coordinate from user space to device space by multiplying the given point by the current transformation matrix (CTM).

Parameters

- **x** (*float*) – X position.
- **y** (*float*) – Y position.

Returns A `(device_x, device_y)` tuple of floats.

user_to_device_distance(*dx*, *dy*)
Transform a distance vector from user space to device space. This method is similar to *Context.user_to_device()* except that the translation components of the CTM will be ignored when transforming `(dx, dy)`.

Parameters

- **dx** – X component of a distance vector.
- **dy** – Y component of a distance vector.

Returns A `(device_dx, device_dy)` tuple of floats.

device_to_user(*x*, *y*)
Transform a coordinate from device space to user space by multiplying the given point by the inverse of the current transformation matrix (CTM).

Parameters

- **x** (*float*) – X position.
- **y** (*float*) – Y position.

Returns A `(user_x, user_y)` tuple of floats.

device_to_user_distance(*dx*, *dy*)
Transform a distance vector from device space to user space. This method is similar to *Context.device_to_user()* except that the translation components of the inverse CTM will be ignored when transforming `(dx, dy)`.

Parameters

- **dx** – X component of a distance vector.
- **dy** – Y component of a distance vector.

Returns A `(user_dx, user_dy)` tuple of floats.

has_current_point()
Returns whether a current point is defined on the current path. See *get_current_point()*.

get_current_point()
Return the current point of the current path, which is conceptually the final point reached by the path so far.

The current point is returned in the user-space coordinate system. If there is no defined current point or if the context is in an error status, `(0, 0)` is returned. It is possible to check this in advance with *has_current_point()*.

Most path construction methods alter the current point. See the following for details on how they affect the current point: *new_path()*, *new_sub_path()*, *append_path()*, *close_path()*, *move_to()*, *line_to()*, *curve_to()*, *rel_move_to()*, *rel_line_to()*, *rel_curve_to()*, *arc()*, *arc_negative()*, *rectangle()*, *text_path()*, *glyph_path()*, `stroke_to_path()`.

Some methods use and alter the current point but do not otherwise change current path: *show_text()*, *show_glyphs()*, *show_text_glyphs()*.

Some methods unset the current path and as a result, current point: *fill()*, *stroke()*.

Returns A `(x, y)` tuple of floats, the coordinates of the current point.

new_path()
Clears the current path. After this call there will be no path and no current point.

new_sub_path()
Begin a new sub-path. Note that the existing path is not affected. After this call there will be no current point.

In many cases, this call is not needed since new sub-paths are frequently started with *move_to()*.

A call to *new_sub_path()* is particularly useful when beginning a new sub-path with one of the *arc()* calls. This makes things easier as it is no longer necessary to manually compute the arc's initial coordinates for a call to *move_to()*.

move_to(*x*, *y*)
Begin a new sub-path. After this call the current point will be `(x, y)`.

Parameters

- **x** – X position of the new point.
- **y** – Y position of the new point.

rel_move_to(*dx*, *dy*)
Begin a new sub-path. After this call the current point will be offset by `(dx, dy)`.

Given a current point of `(x, y)`, `context.rel_move_to(dx, dy)` is logically equivalent to `context.move_to(x + dx, y + dy)`.

Parameters

- **dx** – The X offset.
- **dy** – The Y offset.

Raises *CairoError* if there is no current point. Doing so will cause leave the context in an error state.

line_to(*x*, *y*)

Adds a line to the path from the current point to position `(x, y)` in user-space coordinates. After this call the current point will be `(x, y)`.

If there is no current point before the call to *line_to()* this method will behave as `context.move_to(x, y)`.

Parameters

- **x** – X coordinate of the end of the new line.
- **y** – Y coordinate of the end of the new line.

rel_line_to(*dx*, *dy*)

Relative-coordinate version of *line_to()*. Adds a line to the path from the current point to a point that is offset from the current point by `(dx, dy)` in user space. After this call the current point will be offset by `(dx, dy)`.

Given a current point of `(x, y)`, `context.rel_line_to(dx, dy)` is logically equivalent to `context.line_to(x + dx, y + dy)`.

Parameters

- **dx** – The X offset to the end of the new line.
- **dy** – The Y offset to the end of the new line.

Raises *CairoError* if there is no current point. Doing so will cause leave the context in an error state.

rectangle(*x*, *y*, *width*, *height*)

Adds a closed sub-path rectangle of the given size to the current path at position `(x, y)` in user-space coordinates.

This method is logically equivalent to:

```
context.move_to(x, y)
context.rel_line_to(width, 0)
context.rel_line_to(0, height)
context.rel_line_to(-width, 0)
context.close_path()
```

Parameters

- **x** – The X coordinate of the top left corner of the rectangle.
- **y** – The Y coordinate of the top left corner of the rectangle.
- **width** – Width of the rectangle.
- **height** – Height of the rectangle.

arc(*xc*, *yc*, *radius*, *angle1*, *angle2*)

Adds a circular arc of the given radius to the current path. The arc is centered at `(xc, yc)`, begins at `angle1` and proceeds in the direction of increasing angles to end at `angle2`. If `angle2` is less than `angle1` it will be progressively increased by `2 * pi` until it is greater than `angle1`.

If there is a current point, an initial line segment will be added to the path to connect the current point to the beginning of the arc. If this initial line is undesired, it can be avoided by calling *new_sub_path()* before calling *arc()*.

Angles are measured in radians. An angle of 0 is in the direction of the positive X axis (in user space). An angle of `pi / 2` radians (90 degrees) is in the direction of the positive Y axis (in user space). Angles increase in the direction from the positive X axis toward the positive Y axis. So with the default transformation matrix, angles increase in a clockwise direction.

(To convert from degrees to radians, use `degrees * pi / 180`.)

This method gives the arc in the direction of increasing angles; see *arc_negative()* to get the arc in the direction of decreasing angles.

The arc is circular in user space. To achieve an elliptical arc, you can scale the current transformation matrix by different amounts in the X and Y directions. For example, to draw an ellipse in the box given by x, y, width, height:

```
from math import pi
with context:
    context.translate(x + width / 2., y + height / 2.)
    context.scale(width / 2., height / 2.)
    context.arc(0, 0, 1, 0, 2 * pi)
```

Parameters

- **xc** (*float*) – X position of the center of the arc.
- **yc** (*float*) – Y position of the center of the arc.
- **radius** (*float*) – The radius of the arc.
- **angle1** (*float*) – The start angle, in radians.
- **angle2** (*float*) – The end angle, in radians.

arc_negative(*xc*, *yc*, *radius*, *angle1*, *angle2*)

Adds a circular arc of the given radius to the current path. The arc is centered at `(xc, yc)`, begins at `angle1` and proceeds in the direction of decreasing angles to end at `angle2`. If `angle2` is greater than `angle1` it will be progressively decreased by `2 * pi` until it is greater than `angle1`.

See *arc()* for more details. This method differs only in the direction of the arc between the two angles.

Parameters

- **xc** (*float*) – X position of the center of the arc.
- **yc** (*float*) – Y position of the center of the arc.
- **radius** (*float*) – The radius of the arc.
- **angle1** (*float*) – The start angle, in radians.
- **angle2** (*float*) – The end angle, in radians.

curve_to(*x1*, *y1*, *x2*, *y2*, *x3*, *y3*)

Adds a cubic Bézier spline to the path from the current point to position `(x3, y3)` in user-space coordinates, using `(x1, y1)` and `(x2, y2)` as the control points. After this call the current point will be `(x3, y3)`.

If there is no current point before the call to *curve_to()* this method will behave as if preceded by a call to `context.move_to(x1, y1)`.

Parameters

- **x1** (*float*) – The X coordinate of the first control point.
- **y1** (*float*) – The Y coordinate of the first control point.
- **x2** (*float*) – The X coordinate of the second control point.
- **y2** (*float*) – The Y coordinate of the second control point.
- **x3** (*float*) – The X coordinate of the end of the curve.
- **y3** (*float*) – The Y coordinate of the end of the curve.

rel_curve_to(*dx1*, *dy1*, *dx2*, *dy2*, *dx3*, *dy3*)

Relative-coordinate version of *curve_to()*. All offsets are relative to the current point. Adds a cubic Bézier spline to the path from the current point to a point offset from the current point by `(dx3, dy3)`, using points offset by `(dx1, dy1)` and `(dx2, dy2)` as the control points. After this call the current point will be offset by `(dx3, dy3)`.

Given a current point of `(x, y)`, `context.rel_curve_to(dx1, dy1, dx2, dy2, dx3, dy3)` is logically equivalent to `context.curve_to(x+dx1, y+dy1, x+dx2, y+dy2, x+dx3, y+dy3)`.

Parameters

- **dx1** (*float*) – The X offset to the first control point.
- **dy1** (*float*) – The Y offset to the first control point.
- **dx2** (*float*) – The X offset to the second control point.
- **dy2** (*float*) – The Y offset to the second control point.
- **dx3** (*float*) – The X offset to the end of the curve.
- **dy3** (*float*) – The Y offset to the end of the curve.

Raises *CairoError* if there is no current point. Doing so will cause leave the context in an error state.

text_path(*text*)

Adds closed paths for text to the current path. The generated path if filled, achieves an effect similar to that of *show_text()*.

Text conversion and positioning is done similar to *show_text()*.

Like *show_text()*, after this call the current point is moved to the origin of where the next glyph would be placed in this same progression. That is, the current point will be at the origin of the final glyph offset by its advance values. This allows for chaining multiple calls to to *text_path()* without having to set current point in between.

Parameters **text** – The text to show, as an Unicode or UTF-8 string.

Note: The *text_path()* method is part of what the cairo designers call the "toy" text API. It is convenient for short demos and simple programs, but it is not expected to be adequate for serious text-

using applications. See *Fonts & text* for details, and `glyph_path()` for the "real" text path API in cairo.

---

glyph_path(*glyphs*)
Adds closed paths for the glyphs to the current path. The generated path if filled, achieves an effect similar to that of `show_glyphs()`.

Parameters **glyphs** – The glyphs to show. See `show_text_glyphs()` for the data structure.

close_path()
Adds a line segment to the path from the current point to the beginning of the current sub-path, (the most recent point passed to cairo_move_to()), and closes this sub-path. After this call the current point will be at the joined endpoint of the sub-path.

The behavior of `close_path()` is distinct from simply calling `line_to()` with the equivalent coordinate in the case of stroking. When a closed sub-path is stroked, there are no caps on the ends of the sub-path. Instead, there is a line join connecting the final and initial segments of the sub-path.

If there is no current point before the call to `close_path()`, this method will have no effect.

copy_path()
Return a copy of the current path.

Returns

A list of `(path_operation, coordinates)` tuples of a *Path operation* string and a tuple of floats coordinates whose content depends on the operation type:

- `MOVE_TO`: 1 point `(x, y)`
- `LINE_TO`: 1 point `(x, y)`
- `CURVE_TO`: 3 points `(x1, y1, x2, y2, x3, y3)`
- `CLOSE_PATH` 0 points `()` (empty tuple)

copy_path_flat()
Return a flattened copy of the current path

This method is like `copy_path()` except that any curves in the path will be approximated with piecewise-linear approximations, (accurate to within the current tolerance value, see `set_tolerance()`). That is, the result is guaranteed to not have any elements of type `CURVE_TO` which will instead be replaced by a series of `LINE_TO` elements.

Returns A list of `(path_operation, coordinates)` tuples. See `copy_path()` for the data structure.

append_path(*path*)
Append `path` onto the current path. The path may be either the return value from one of `copy_path()` or `copy_path_flat()` or it may be constructed manually.

Parameters **path** – An iterable of tuples in the same format as returned by `copy_path()`.

path_extents()
Computes a bounding box in user-space coordinates covering the points on the current path. If the current path is empty, returns an empty rectangle `(0, 0, 0, 0)`. Stroke parameters, fill rule, surface dimensions and clipping are not taken into account.

Contrast with `fill_extents()` and `stroke_extents()` which return the extents of only the area that would be "inked" by the corresponding drawing operations.

The result of *path_extents()* is defined as equivalent to the limit of *stroke_extents()* with *LINE_CAP_ROUND* as the line width approaches 0, (but never reaching the empty-rectangle returned by *stroke_extents()* for a line width of 0).

Specifically, this means that zero-area sub-paths such as *move_to()*; *line_to()* segments, (even degenerate cases where the coordinates to both calls are identical), will be considered as contributing to the extents. However, a lone *move_to()* will not contribute to the results of *path_extents()*.

Returns A `(x1, y1, x2, y2)` tuple of floats: the left, top, right and bottom of the resulting extents, respectively.

paint()
A drawing operator that paints the current source everywhere within the current clip region.

paint_with_alpha(*alpha*)
A drawing operator that paints the current source everywhere within the current clip region using a mask of constant alpha value alpha. The effect is similar to *paint()*, but the drawing is faded out using the `alpha` value.

Parameters **alpha** (*float*) – Alpha value, between 0 (transparent) and 1 (opaque).

mask(*pattern*)
A drawing operator that paints the current source using the alpha channel of `pattern` as a mask. (Opaque areas of `pattern` are painted with the source, transparent areas are not painted.)

Parameters **pattern** – A *Pattern* object.

mask_surface(*surface*, *surface_x=0*, *surface_y=0*)
A drawing operator that paints the current source using the alpha channel of `surface` as a mask. (Opaque areas of `surface` are painted with the source, transparent areas are not painted.)

Parameters

- **pattern** – A *Surface* object.
- **surface_x** (*float*) – X coordinate at which to place the origin of surface.
- **surface_y** (*float*) – Y coordinate at which to place the origin of surface.

fill()
A drawing operator that fills the current path according to the current fill rule, (each sub-path is implicitly closed before being filled). After *fill()*, the current path will be cleared from the cairo context.

See *set_fill_rule()* and *fill_preserve()*.

fill_preserve()
A drawing operator that fills the current path according to the current fill rule, (each sub-path is implicitly closed before being filled). Unlike *fill()*, *fill_preserve()* preserves the path within the cairo context.

See *set_fill_rule()* and *fill()*.

fill_extents()
Computes a bounding box in user-space coordinates covering the area that would be affected, (the "inked" area), by a *fill()* operation given the current path and fill parameters. If the current path is empty, returns an empty rectangle `(0, 0, 0, 0)`. Surface dimensions and clipping are not taken into account.

Contrast with *path_extents()* which is similar, but returns non-zero extents for some paths with no inked area, (such as a simple line segment).

Note that *fill_extents()* must necessarily do more work to compute the precise inked areas in light of the fill rule, so *path_extents()* may be more desirable for sake of performance if the non-inked path extents are desired.

See *fill()*, *set_fill_rule()* and *fill_preserve()*.

Returns A `(x1, y1, x2, y2)` tuple of floats: the left, top, right and bottom of the resulting extents, respectively.

in_fill(*x*, *y*)

Tests whether the given point is inside the area that would be affected by a *fill()* operation given the current path and filling parameters. Surface dimensions and clipping are not taken into account.

See *fill()*, *set_fill_rule()* and *fill_preserve()*.

Parameters

- **x** (*float*) – X coordinate of the point to test
- **y** (*float*) – Y coordinate of the point to test

Returns A boolean.

stroke()

A drawing operator that strokes the current path according to the current line width, line join, line cap, and dash settings. After *stroke()*, the current path will be cleared from the cairo context. See *set_line_width()*, *set_line_join()*, *set_line_cap()*, *set_dash()*, and *stroke_preserve()*.

Note: Degenerate segments and sub-paths are treated specially and provide a useful result. These can result in two different situations:

1. Zero-length "on" segments set in *set_dash()*. If the cap style is *ROUND* or *SQUARE* then these segments will be drawn as circular dots or squares respectively. In the case of *SQUARE*, the orientation of the squares is determined by the direction of the underlying path.
2. A sub-path created by *move_to()* followed by either a *close_path()* or one or more calls to *line_to()* to the same coordinate as the *move_to()*. If the cap style is *ROUND* then these sub-paths will be drawn as circular dots. Note that in the case of *SQUARE* a degenerate sub-path will not be drawn at all, (since the correct orientation is indeterminate).

In no case will a cap style of *BUTT* cause anything to be drawn in the case of either degenerate segments or sub-paths.

stroke_preserve()

A drawing operator that strokes the current path according to the current line width, line join, line cap, and dash settings. Unlike *stroke()*, *stroke_preserve()* preserves the path within the cairo context. See *set_line_width()*, *set_line_join()*, *set_line_cap()*, *set_dash()*, and *stroke()*.

stroke_extents()

Computes a bounding box in user-space coordinates covering the area that would be affected, (the "inked" area), by a *stroke()* operation given the current path and stroke parameters. If the current path is empty, returns an empty rectangle `(0, 0, 0, 0)`. Surface dimensions and clipping are not taken into account.

Note that if the line width is set to exactly zero, then *stroke_extents()* will return an empty rectangle. Contrast with *path_extents()* which can be used to compute the non-empty bounds as the line width approaches zero.

Note that *stroke_extents()* must necessarily do more work to compute the precise inked areas in light of the stroke parameters, so *path_extents()* may be more desirable for sake of performance if the non-inked path extents are desired.

See *stroke()*, *set_line_width()*, *set_line_join()*, *set_line_cap()*, *set_dash()*, and *stroke_preserve()*.

Returns A `(x1, y1, x2, y2)` tuple of floats: the left, top, right and bottom of the resulting extents, respectively.

in_stroke(*x*, *y*)

Tests whether the given point is inside the area that would be affected by a *stroke()* operation given the current path and stroking parameters. Surface dimensions and clipping are not taken into account.

See *stroke()*, *set_line_width()*, *set_line_join()*, *set_line_cap()*, *set_dash()*, and *stroke_preserve()*.

Parameters

- **x** (*float*) – X coordinate of the point to test
- **y** (*float*) – Y coordinate of the point to test

Returns A boolean.

clip()

Establishes a new clip region by intersecting the current clip region with the current path as it would be filled by *fill()* and according to the current fill rule (see *set_fill_rule()*).

After *clip()*, the current path will be cleared from the cairo context.

The current clip region affects all drawing operations by effectively masking out any changes to the surface that are outside the current clip region.

Calling *clip()* can only make the clip region smaller, never larger. But the current clip is part of the graphics state, so a temporary restriction of the clip region can be achieved by calling *clip()* within a *save()* / *restore()* pair. The only other means of increasing the size of the clip region is *reset_clip()*.

clip_preserve()

Establishes a new clip region by intersecting the current clip region with the current path as it would be filled by *fill()* and according to the current fill rule (see *set_fill_rule()*).

Unlike *clip()*, *clip_preserve()* preserves the path within the cairo context.

The current clip region affects all drawing operations by effectively masking out any changes to the surface that are outside the current clip region.

Calling *clip_preserve()* can only make the clip region smaller, never larger. But the current clip is part of the graphics state, so a temporary restriction of the clip region can be achieved by calling *clip_preserve()* within a *save()* / *restore()* pair. The only other means of increasing the size of the clip region is *reset_clip()*.

clip_extents()

Computes a bounding box in user coordinates covering the area inside the current clip.

Returns A `(x1, y1, x2, y2)` tuple of floats: the left, top, right and bottom of the resulting extents, respectively.

copy_clip_rectangle_list()

Return the current clip region as a list of rectangles in user coordinates.

Returns A list of rectangles, as `(x, y, width, height)` tuples of floats.

Raises *CairoError* if the clip region cannot be represented as a list of user-space rectangles.

in_clip(*x*, *y*)

Tests whether the given point is inside the area that would be visible through the current clip, i.e. the area that would be filled by a *paint()* operation.

See *clip()*, and *clip_preserve()*.

Parameters

- **x** (*float*) – X coordinate of the point to test
- **y** (*float*) – Y coordinate of the point to test

Returns A boolean.

New in cairo 1.10.

reset_clip()
Reset the current clip region to its original, unrestricted state. That is, set the clip region to an infinitely large shape containing the target surface. Equivalently, if infinity is too hard to grasp, one can imagine the clip region being reset to the exact bounds of the target surface.

Note that code meant to be reusable should not call *reset_clip()* as it will cause results unexpected by higher-level code which calls *clip()*. Consider using *cairo()* and *restore()* around *clip()* as a more robust means of temporarily restricting the clip region.

select_font_face(*family=''*, *slant=0*, *weight=0*)
Selects a family and style of font from a simplified description as a family name, slant and weight.

Note: The *select_font_face()* method is part of what the cairo designers call the "toy" text API. It is convenient for short demos and simple programs, but it is not expected to be adequate for serious text-using applications. See *Fonts & text* for details.

Cairo provides no operation to list available family names on the system (this is a "toy", remember), but the standard CSS2 generic family names, (`"serif"`, `"sans-serif"`, `"cursive"`, `"fantasy"`, `"monospace"`), are likely to work as expected.

If family starts with the string `"cairo:"`, or if no native font backends are compiled in, cairo will use an internal font family. The internal font family recognizes many modifiers in the family string, most notably, it recognizes the string `"monospace"`. That is, the family name `"cairo:monospace"` will use the monospace version of the internal font family.

If text is drawn without a call to *select_font_face()*, (nor *set_font_face()* nor *set_scaled_font()*), the default family is platform-specific, but is essentially `"sans-serif"`. Default slant is *NORMAL*, and default weight is *NORMAL*.

This method is equivalent to a call to *ToyFontFace* followed by *set_font_face()*.

set_font_face(*font_face*)
Replaces the current font face with `font_face`.

Parameters **font_face** – A *FontFace* object, or `None` to restore the default font.

get_font_face()
Return the current font face.

Parameters **font_face** – A new *FontFace* object wrapping an existing cairo object.

set_font_size(*size*)
Sets the current font matrix to a scale by a factor of `size`, replacing any font matrix previously set with *set_font_size()* or *set_font_matrix()*. This results in a font size of size user space units. (More precisely, this matrix will result in the font's em-square being a size by size square in user space.)

If text is drawn without a call to *set_font_size()*, (nor *set_font_matrix()* nor *set_scaled_font()*), the default font size is 10.0.

Parameters **size** (*float*) – The new font size, in user space units

set_font_matrix(*matrix*)
Sets the current font matrix to `matrix`. The font matrix gives a transformation from the design space of the font (in this space, the em-square is 1 unit by 1 unit) to user space. Normally, a simple scale is used (see *set_font_size()*), but a more complex font matrix can be used to shear the font or stretch it unequally along the two axes

Parameters **matrix** – A *Matrix* describing a transform to be applied to the current font.

get_font_matrix()
Copies the current font matrix. See *set_font_matrix()*.

Returns A new *Matrix*.

set_font_options(*font_options*)
Sets a set of custom font rendering options. Rendering options are derived by merging these options with the options derived from underlying surface; if the value in options has a default value (like *ANTIALIAS_DEFAULT*), then the value from the surface is used.

Parameters **font_options** – A *FontOptions* object.

get_font_options()
Retrieves font rendering options set via *set_font_options()*. Note that the returned options do not include any options derived from the underlying surface; they are literally the options passed to *set_font_options()*.

Returns A new *FontOptions* object.

set_scaled_font(*scaled_font*)
Replaces the current font face, font matrix, and font options with those of `scaled_font`. Except for some translation, the current CTM of the context should be the same as that of the `scaled_font`, which can be accessed using *ScaledFont.get_ctm()*.

Parameters **scaled_font** – A *ScaledFont* object.

get_scaled_font()
Return the current scaled font.

Returns A new *ScaledFont* object, wrapping an existing cairo object.

font_extents()
Return the extents of the currently selected font.

Values are given in the current user-space coordinate system.

Because font metrics are in user-space coordinates, they are mostly, but not entirely, independent of the current transformation matrix. If you call *context.scale(2)*, text will be drawn twice as big, but the reported text extents will not be doubled. They will change slightly due to hinting (so you can't assume that metrics are independent of the transformation matrix), but otherwise will remain unchanged.

Returns A `(ascent, descent, height, max_x_advance, max_y_advance)` tuple of floats.

ascent The distance that the font extends above the baseline. Note that this is not always exactly equal to the maximum of the extents of all the glyphs in the font, but rather is picked to express the font designer's intent as to how the font should align with elements above it.

descent The distance that the font extends below the baseline. This value is positive for typical fonts that include portions below the baseline. Note that this is not always exactly equal to the maximum of the extents of all the glyphs in the font, but rather is picked to express the font designer's intent as to how the font should align with elements below it.

height The recommended vertical distance between baselines when setting consecutive lines of text with the font. This is greater than `ascent + descent` by a quantity known as the line spacing or external leading. When space is at a premium, most fonts can be set with only a distance of `ascent + descent` between lines.

max_x_advance The maximum distance in the X direction that the origin is advanced for any glyph in the font.

max_y_advance The maximum distance in the Y direction that the origin is advanced for any glyph in the font. This will be zero for normal fonts used for horizontal writing. (The scripts of East Asia are sometimes written vertically.)

text_extents(*text*)

Returns the extents for a string of text.

The extents describe a user-space rectangle that encloses the "inked" portion of the text, (as it would be drawn by *show_text()*). Additionally, the `x_advance` and `y_advance` values indicate the amount by which the current point would be advanced by *show_text()*.

Note that whitespace characters do not directly contribute to the size of the rectangle (`width` and `height`). They do contribute indirectly by changing the position of non-whitespace characters. In particular, trailing whitespace characters are likely to not affect the size of the rectangle, though they will affect the x_advance and y_advance values.

Because text extents are in user-space coordinates, they are mostly, but not entirely, independent of the current transformation matrix. If you call *context.scale(2)*, text will be drawn twice as big, but the reported text extents will not be doubled. They will change slightly due to hinting (so you can't assume that metrics are independent of the transformation matrix), but otherwise will remain unchanged.

Parameters **text** – The text to measure, as an Unicode or UTF-8 string.

Returns A `(x_bearing, y_bearing, width, height, x_advance, y_advance)` tuple of floats.

x_bearing The horizontal distance from the origin to the leftmost part of the glyphs as drawn. Positive if the glyphs lie entirely to the right of the origin.

y_bearing The vertical distance from the origin to the topmost part of the glyphs as drawn. Positive only if the glyphs lie completely below the origin; will usually be negative.

width Width of the glyphs as drawn.

height Height of the glyphs as drawn.

x_advance Distance to advance in the X direction after drawing these glyphs.

y_advance Distance to advance in the Y direction after drawing these glyphs. Will typically be zero except for vertical text layout as found in East-Asian languages.

glyph_extents(*glyphs*)

Returns the extents for a list of glyphs.

The extents describe a user-space rectangle that encloses the "inked" portion of the glyphs, (as it would be drawn by *show_glyphs()*). Additionally, the `x_advance` and `y_advance` values indicate the amount by which the current point would be advanced by *show_glyphs()*.

Parameters **glyphs** – A list of glyphs. See *show_text_glyphs()* for the data structure.

Returns A `(x_bearing, y_bearing, width, height, x_advance, y_advance)` tuple of floats. See *text_extents()* for details.

show_text(*text*)

A drawing operator that generates the shape from a string text, rendered according to the current font *face*, font *size* (font *matrix*), and font *options*.

This method first computes a set of glyphs for the string of text. The first glyph is placed so that its origin is at the current point. The origin of each subsequent glyph is offset from that of the previous glyph by the advance values of the previous glyph.

After this call the current point is moved to the origin of where the next glyph would be placed in this same progression. That is, the current point will be at the origin of the final glyph offset by its advance values. This allows for easy display of a single logical string with multiple calls to *show_text()*.

Parameters **text** – The text to show, as an Unicode or UTF-8 string.

Note: This method is part of what the cairo designers call the "toy" text API. It is convenient for short demos and simple programs, but it is not expected to be adequate for serious text-using applications. See *Fonts & text* for details and *show_glyphs()* for the "real" text display API in cairo.

---

show_glyphs(*glyphs*)

A drawing operator that generates the shape from a list of glyphs, rendered according to the current font *face*, font *size* (font *matrix*), and font *options*.

Parameters **glyphs** – The glyphs to show. See *show_text_glyphs()* for the data structure.

show_text_glyphs(*text*, *glyphs*, *clusters*, *cluster_flags=0*)

This operation has rendering effects similar to *show_glyphs()* but, if the target surface supports it (see *Surface.has_show_text_glyphs()*), uses the provided text and cluster mapping to embed the text for the glyphs shown in the output. If the target does not support the extended attributes, this method acts like the basic *show_glyphs()* as if it had been passed `glyphs`.

The mapping between `text` and `glyphs` is provided by an list of clusters. Each cluster covers a number of UTF-8 text bytes and glyphs, and neighboring clusters cover neighboring areas of `text` and `glyphs`. The clusters should collectively cover `text` and `glyphs` in entirety.

Parameters

- **text** – The text to show, as an Unicode or UTF-8 string. Because of how `clusters` work, using UTF-8 bytes might be more convenient.
- **glyphs** – A list of glyphs. Each glyph is a `(glyph_id, x, y)` tuple. `glyph_id` is an opaque integer. Its exact interpretation depends on the font technology being used. `x` and `y` are the float offsets in the X and Y direction between the origin used for drawing or measuring the string and the origin of this glyph. Note that the offsets are not cumulative. When drawing or measuring text, each glyph is individually positioned with respect to the overall origin.
- **clusters** – A list of clusters. A text cluster is a minimal mapping of some glyphs corresponding to some UTF-8 text, represented as a `(num_bytes, num_glyphs)` tuple of integers, the number of UTF-8 bytes and glyphs covered by the cluster. For a cluster to be valid, both `num_bytes` and `num_glyphs` should be non-negative, and at least one should be non-zero. Note that clusters with zero glyphs are not as well supported as normal clusters. For example, PDF rendering applications typically ignore those clusters when PDF text is being selected.
- **cluster_flags** (*int*) – Flags (as a bit field) for the cluster mapping. The first cluster always covers bytes from the beginning of `text`. If `cluster_flags` does not have the *TEXT_CLUSTER_FLAG_BACKWARD* flag set, the first cluster also covers the beginning

of `glyphs`, otherwise it covers the end of the `glyphs` list and following clusters move backward.

show_page()
Emits and clears the current page for backends that support multiple pages. Use *copy_page()* if you don't want to clear the page.

This is a convenience method that simply calls *Surface.show_page()* on the context's target.

copy_page()
Emits the current page for backends that support multiple pages, but doesn't clear it, so the contents of the current page will be retained for the next page too. Use *show_page()* if you want to clear the page.

This is a convenience method that simply calls *Surface.copy_page()* on the context's target.

Matrix

class cairocffi.**Matrix**(*xx=1*, *yx=0*, *xy=0*, *yy=1*, *x0=0*, *y0=0*)
A 2D transformation matrix.

Matrices are used throughout cairo to convert between different coordinate spaces. A *Matrix* holds an affine transformation, such as a scale, rotation, shear, or a combination of these. The transformation of a point (x,y) is given by:

```
x_new = xx * x + xy * y + x0
y_new = yx * x + yy * y + y0
```

The current transformation matrix of a *Context*, represented as a *Matrix*, defines the transformation from user-space coordinates to device-space coordinates. See *Context.get_matrix()* and *Context.set_matrix()*.

The default values produce an identity matrix.

Matrices can be compared with `m1 == m2` and `m2 != m2` as well as multiplied with `m3 = m1 * m2`.

classmethod init_rotate(*radians*)
Return a new *Matrix* for a transformation that rotates by `radians`.

Parameters **radians** (*float*) – Angle of rotation, in radians. The direction of rotation is defined such that positive angles rotate in the direction from the positive X axis toward the positive Y axis. With the default axis orientation of cairo, positive angles rotate in a clockwise direction.

as_tuple()
Return all of the matrix's components.

Returns A `(xx, yx, xy, yy, x0, y0)` tuple of floats.

copy()
Return a new copy of this matrix.

multiply(*other*)
Multiply with another matrix and return the result as a new *Matrix* object. Same as `self * other`.

translate(*tx*, *ty*)
Applies a translation by `tx`, `ty` to the transformation in this matrix.

The effect of the new transformation is to first translate the coordinates by `tx` and `ty`, then apply the original transformation to the coordinates.

Note: This changes the matrix in-place.

Parameters

- **tx** (*float*) – Amount to translate in the X direction.
- **ty** (*float*) – Amount to translate in the Y direction.

scale(*sx*, *sy=None*)

Applies scaling by `sx`, `sy` to the transformation in this matrix.

The effect of the new transformation is to first scale the coordinates by `sx` and `sy`, then apply the original transformation to the coordinates.

If `sy` is omitted, it is the same as `sx` so that scaling preserves aspect ratios.

Note: This changes the matrix in-place.

Parameters

- **sx** (*float*) – Scale factor in the X direction.
- **sy** (*float*) – Scale factor in the Y direction.

rotate(*radians*)

Applies a rotation by `radians` to the transformation in this matrix.

The effect of the new transformation is to first rotate the coordinates by `radians`, then apply the original transformation to the coordinates.

Note: This changes the matrix in-place.

Parameters **radians** (*float*) – Angle of rotation, in radians. The direction of rotation is defined such that positive angles rotate in the direction from the positive X axis toward the positive Y axis. With the default axis orientation of cairo, positive angles rotate in a clockwise direction.

invert()

Changes matrix to be the inverse of its original value. Not all transformation matrices have inverses; if the matrix collapses points together (it is degenerate), then it has no inverse and this function will fail.

Note: This changes the matrix in-place.

Raises *CairoError* on degenerate matrices.

inverted()

Return the inverse of this matrix. See *invert()*.

Raises *CairoError* on degenerate matrices.

Returns A new *Matrix* object.

transform_point(*x*, *y*)
Transforms the point `(x, y)` by this matrix.

Parameters

- **x** (*float*) – X position.
- **y** (*float*) – Y position.

Returns A `(new_x, new_y)` tuple of floats.

transform_distance(*dx*, *dy*)
Transforms the distance vector `(dx, dy)` by this matrix. This is similar to *transform_point()* except that the translation components of the transformation are ignored. The calculation of the returned vector is as follows:

```
dx2 = dx1 * xx + dy1 * xy
dy2 = dx1 * yx + dy1 * yy
```

Affine transformations are position invariant, so the same vector always transforms to the same vector. If `(x1, y1)` transforms to `(x2, y2)` then `(x1 + dx1, y1 + dy1)` will transform to `(x1 + dx2, y1 + dy2)` for all values of `x1` and `x2`.

Parameters

- **dx** (*float*) – X component of a distance vector.
- **dy** (*float*) – Y component of a distance vector.

Returns A `(new_dx, new_dy)` tuple of floats.

xx
Read-write attribute access to a single float component.

yx
Read-write attribute access to a single float component.

xy
Read-write attribute access to a single float component.

yy
Read-write attribute access to a single float component.

x0
Read-write attribute access to a single float component.

y0
Read-write attribute access to a single float component.

Patterns

class cairocffi.**Pattern**
The base class for all pattern types.

Should not be instantiated directly, but see *CFFI API*. An instance may be returned for cairo pattern types that are not (yet) defined in cairocffi.

A *Pattern* represents a source when drawing onto a surface. There are different sub-classes of *Pattern*, for different types of sources; for example, *SolidPattern* is a pattern for a solid color.

Other than instantiating the various *Pattern* sub-classes, some of the pattern types can be implicitly created using various *Context*; for example *Context.set_source_rgb()*.

set_extend(*extend*)
Sets the mode to be used for drawing outside the area of this pattern. See *Pattern extend* for details on the semantics of each extend strategy.

The default extend mode is *NONE* for *SurfacePattern* and *PAD* for *Gradient* patterns.

get_extend()
Gets the current extend mode for this pattern.

Returns A *Pattern extend* string.

set_filter(*filter*)
Sets the filter to be used for resizing when using this pattern. See *Pixel filter* for details on each filter.

Note that you might want to control filtering even when you do not have an explicit *Pattern*, (for example when using *Context.set_source_surface()*). In these cases, it is convenient to use *Context.get_source()* to get access to the pattern that cairo creates implicitly.

For example:

```
context.get_source().set_filter(cairocffi.FILTER_NEAREST)
```

get_filter()
Return the current filter string for this pattern. See *Pixel filter* for details on each filter.

set_matrix(*matrix*)
Sets the pattern's transformation matrix to `matrix`. This matrix is a transformation from user space to pattern space.

When a pattern is first created it always has the identity matrix for its transformation matrix, which means that pattern space is initially identical to user space.

Important: Please note that the direction of this transformation matrix is from user space to pattern space. This means that if you imagine the flow from a pattern to user space (and on to device space), then coordinates in that flow will be transformed by the inverse of the pattern matrix.

For example, if you want to make a pattern appear twice as large as it does by default the correct code to use is:

```
pattern.set_matrix(Matrix(xx=0.5, yy=0.5))
```

Meanwhile, using values of 2 rather than 0.5 in the code above would cause the pattern to appear at half of its default size.

Also, please note the discussion of the user-space locking semantics of *Context.set_source()*.

Parameters **matrix** – A *Matrix* to be copied into the pattern.

get_matrix()
Copies the pattern's transformation matrix.

Retuns A new *Matrix* object.

SolidPattern

class cairocffi.**SolidPattern**(*red*, *green*, *blue*, *alpha=1*)
Creates a new pattern corresponding to a solid color. The color and alpha components are in the range 0 to 1. If the values passed in are outside that range, they will be clamped.

Parameters

- **red** (*float*) – Red component of the color.

- **green** (*float*) – Green component of the color.
- **blue** (*float*) – Blue component of the color.
- **alpha** (*float*) – Alpha component of the color. 1 (the default) is opaque, 0 fully transparent.

get_rgba()
Returns the solid pattern's color.

Returns a `(red, green, blue, alpha)` tuple of floats.

SurfacePattern

class cairocffi.**SurfacePattern**(*surface*)
Create a new pattern for the given surface.

Parameters **surface** – A *Surface* object.

get_surface()
Return this *SurfacePattern*'s surface.

Returns An instance of *Surface* or one of its sub-classes, a new Python object referencing the existing cairo surface.

Gradient

class cairocffi.**Gradient**
The common parent of *LinearGradient* and *RadialGradient*. Should not be instantiated directly.

add_color_stop_rgba(*offset*, *red*, *green*, *blue*, *alpha=1*)
Adds a translucent color stop to a gradient pattern.

The offset specifies the location along the gradient's control vector. For example, a linear gradient's control vector is from (x0,y0) to (x1,y1) while a radial gradient's control vector is from any point on the start circle to the corresponding point on the end circle.

If two (or more) stops are specified with identical offset values, they will be sorted according to the order in which the stops are added (stops added earlier before stops added later). This can be useful for reliably making sharp color transitions instead of the typical blend.

The color components and offset are in the range 0 to 1. If the values passed in are outside that range, they will be clamped.

Parameters

- **offset** (*float*) – Location along the gradient's control vector
- **red** (*float*) – Red component of the color.
- **green** (*float*) – Green component of the color.
- **blue** (*float*) – Blue component of the color.
- **alpha** (*float*) – Alpha component of the color. 1 (the default) is opaque, 0 fully transparent.

add_color_stop_rgb(*offset*, *red*, *green*, *blue*)
Same as *add_color_stop_rgba()* with `alpha=1`. Kept for compatibility with pycairo.

get_color_stops()
Return this gradient's color stops so far.

Returns A list of `(offset, red, green, blue, alpha)` tuples of floats.

LinearGradient

class `cairocffi.`**`LinearGradient`**(*x0*, *y0*, *x1*, *y1*)

Create a new linear gradient along the line defined by (x0, y0) and (x1, y1). Before using the gradient pattern, a number of color stops should be defined using *add_color_stop_rgba()*.

Note: The coordinates here are in pattern space. For a new pattern, pattern space is identical to user space, but the relationship between the spaces can be changed with *set_matrix()*.

Parameters

- **x0** (*float*) – X coordinate of the start point.
- **y0** (*float*) – Y coordinate of the start point.
- **x1** (*float*) – X coordinate of the end point.
- **y1** (*float*) – Y coordinate of the end point.

get_linear_points()

Return this linear gradient's endpoints.

Returns A `(x0, y0, x1, y1)` tuple of floats.

RadialGradient

class `cairocffi.`**`RadialGradient`**(*cx0*, *cy0*, *radius0*, *cx1*, *cy1*, *radius1*)

Creates a new radial gradient pattern between the two circles defined by (cx0, cy0, radius0) and (cx1, cy1, radius1). Before using the gradient pattern, a number of color stops should be defined using *add_color_stop_rgba()*.

Note: The coordinates here are in pattern space. For a new pattern, pattern space is identical to user space, but the relationship between the spaces can be changed with *set_matrix()*.

Parameters

- **cx0** (*float*) – X coordinate of the start circle.
- **cy0** (*float*) – Y coordinate of the start circle.
- **radius0** (*float*) – Radius of the start circle.
- **cx1** (*float*) – X coordinate of the end circle.
- **cy1** (*float*) – Y coordinate of the end circle.
- **radius1** (*float*) – Y coordinate of the end circle.

get_radial_circles()

Return this radial gradient's endpoint circles, each specified as a center coordinate and a radius.

Returns A `(cx0, cy0, radius0, cx1, cy1, radius1)` tuple of floats.

Fonts & text

A font is (in simple terms) a collection of shapes used to draw text. A *glyph* is one of these shapes. There can be multiple glyphs for the same character (alternates to be used in different contexts, for example), or a glyph can be a ligature of multiple characters. Converting text to positioned glyphs is *shaping*.

Cairo itself provides a "toy" text API that only does simple shaping: no ligature or kerning; one glyph per character, positioned by moving the cursor by the X and Y *advance* of each glyph.

It is expected that most applications will need to *use Pango* or a similar library in conjunction with cairo for more comprehensive font handling and text layout.

Font faces

Note: At the moment cairocffi only supports cairo's "toy" font selection API. `FontFace` objects of other types could be obtained eg. from `Context.get_font_face()`, but they can not be instantiated directly.

---

class cairocffi.**FontFace**
The base class for all font face types.

Should not be instantiated directly, but see *CFFI API*. An instance may be returned for cairo font face types that are not (yet) defined in cairocffi.

ToyFontFace

class cairocffi.**ToyFontFace**(*family=''*, *slant=0*, *weight=0*)
Creates a font face from a triplet of family, slant, and weight. These font faces are used in implementation of cairo's "toy" font API.

If family is the zero-length string `""`, the platform-specific default family is assumed. The default family then can be queried using `get_family()`.

The `Context.select_font_face()` method uses this to create font faces. See that method for limitations and other details of toy font faces.

Parameters

- **family** – a font family name, as an Unicode or UTF-8 string.
- **slant** – The *Font slant* string for the font face.
- **weight** – The *Font weight* string for the font face.

get_family()
Return this font face's family name.

get_slant()
Return this font face's *Font slant* string.

get_weight()
Return this font face's *Font weight* string.

ScaledFont

class cairocffi.**ScaledFont**(*font_face*, *font_matrix=None*, *ctm=None*, *options=None*)
Creates a `ScaledFont` object from a font face and matrices that describe the size of the font and the environment in which it will be used.

Parameters

- **font_face** – A `FontFace` object.

- **font_matrix** (Matrix) – Font space to user space transformation matrix for the font. In the simplest case of a N point font, this matrix is just a scale by N, but it can also be used to shear the font or stretch it unequally along the two axes. If omitted, a scale by 10 matrix is assumed (ie. a 10 point font size). See *Context.set_font_matrix*.
- **ctm** (Matrix) – User to device transformation matrix with which the font will be used. If omitted, an identity matrix is assumed.
- **options** – The *FontOptions* object to use when getting metrics for the font and rendering with it. If omitted, the default options are assumed.

get_font_face()
Return the font face that this scaled font uses.

Returns A new instance of *FontFace* (or one of its sub-classes). Might wrap be the same font face passed to *ScaledFont*, but this does not hold true for all possible cases.

get_font_options()
Copies the scaled font's options.

Returns A new *FontOptions* object.

get_font_matrix()
Copies the scaled font's font matrix.

Returns A new *Matrix* object.

get_ctm()
Copies the scaled font's font current transform matrix.

Note that the translation offsets `(x0, y0)` of the CTM are ignored by *ScaledFont*. So, the matrix this method returns always has 0 as `x0` and `y0`.

Returns A new *Matrix* object.

get_scale_matrix()
Copies the scaled font's scaled matrix.

The scale matrix is product of the font matrix and the ctm associated with the scaled font, and hence is the matrix mapping from font space to device space.

Returns A new *Matrix* object.

extents()
Return the scaled font's extents. See *Context.font_extents()*.

Returns A `(ascent, descent, height, max_x_advance, max_y_advance)` tuple of floats.

text_extents(*text*)
Returns the extents for a string of text.

The extents describe a user-space rectangle that encloses the "inked" portion of the text, (as it would be drawn by `show_text()`). Additionally, the `x_advance` and `y_advance` values indicate the amount by which the current point would be advanced by `show_text()`.

Parameters **text** – The text to measure, as an Unicode or UTF-8 string.

Returns A `(x_bearing, y_bearing, width, height, x_advance, y_advance)` tuple of floats. See *Context.text_extents()* for details.

glyph_extents(*glyphs*)
Returns the extents for a list of glyphs.

The extents describe a user-space rectangle that encloses the "inked" portion of the glyphs, (as it would be drawn by `show_glyphs()`). Additionally, the `x_advance` and `y_advance` values indicate the amount by which the current point would be advanced by `show_glyphs()`.

Parameters **glyphs** – A list of glyphs, as returned by *text_to_glyphs()*. Each glyph is a `(glyph_id, x, y)` tuple of an integer and two floats.

Returns A `(x_bearing, y_bearing, width, height, x_advance, y_advance)` tuple of floats. See *Context.text_extents()* for details.

text_to_glyphs (*x*, *y*, *text*, *with_clusters*)

Converts a string of text to a list of glyphs, optionally with cluster mapping, that can be used to render later using this scaled font.

The output values can be readily passed to *Context.show_text_glyphs()*, *Context.show_glyphs()* or related methods, assuming that the exact same *ScaledFont* is used for the operation.

Parameters

- **x** (*float*) – X position to place first glyph.
- **y** (*float*) – Y position to place first glyph.
- **text** – The text to convert, as an Unicode or UTF-8 string.
- **with_clusters** (*bool*) – Whether to compute the cluster mapping.

Returns A `(glyphs, clusters, clusters_flags)` tuple if `with_clusters` is true, otherwise just `glyphs`. See *Context.show_text_glyphs()* for the data structure.

Note: This method is part of what the cairo designers call the "toy" text API. It is convenient for short demos and simple programs, but it is not expected to be adequate for serious text-using applications. See *Fonts & text* for details and *Context.show_glyphs()* for the "real" text display API in cairo.

FontOptions

class cairocffi.**FontOptions**(***values*)

An opaque object holding all options that are used when rendering fonts.

Individual features of a *FontOptions* can be set or accessed using method named `set_FEATURE_NAME()` and `get_FEATURE_NAME()`, like *set_antialias()* and *get_antialias()*.

New features may be added to *FontOptions* in the future. For this reason, ==, *copy()*, *merge()*, and `hash()` should be used to check for equality copy,, merge, or compute a hash value of *FontOptions* objects.

Parameters **values** – Call the corresponding `set_FEATURE_NAME()` methods after creating a new *FontOptions*:

```
options = FontOptions()
options.set_antialias(cairocffi.ANTIALIAS_BEST)
assert FontOptions(antialias=cairocffi.ANTIALIAS_BEST) == options
```

copy()

Return a new *FontOptions* with the same values.

merge(*other*)
Merges non-default options from `other`, replacing existing values. This operation can be thought of as somewhat similar to compositing other onto options with the operation of *OVER*.

set_antialias(*antialias*)
Changes the *Antialiasing mode* for the font options object. This specifies the type of antialiasing to do when rendering text.

get_antialias()
Return the *Antialiasing mode* string for the font options object.

set_subpixel_order(*subpixel_order*)
Changes the *Subpixel order* for the font options object. The subpixel order specifies the order of color elements within each pixel on the display device when rendering with an antialiasing mode of *SUBPIXEL*.

get_subpixel_order()
Return the *Subpixel order* string for the font options object.

set_hint_style(*hint_style*)
Changes the *Hint style* for the font options object. This controls whether to fit font outlines to the pixel grid, and if so, whether to optimize for fidelity or contrast.

get_hint_style()
Return the *Hint style* string for the font options object.

set_hint_metrics(*hint_metrics*)
Changes the *Metrics hinting mode* for the font options object. This controls whether metrics are quantized to integer values in device units.

get_hint_metrics()
Return the *Metrics hinting mode* string for the font options object.

Enumerated values

Some parameters or return values in the cairo API only have a fixed, finite set of valid values. These are represented as *enumerated types* in C, and as integers in CFFI. Users are encouraged to use the constants defined here in the *cairocffi* module rather than literal integers. For example:

```
surface = cairocffi.ImageSurface(cairocffi.FORMAT_ARGB32, 300, 400)
```

Content

Used to describe the content that a *Surface* will contain, whether color information, alpha information (translucence vs. opacity), or both.

cairocffi.**CONTENT_COLOR**
The surface will hold color content only.

cairocffi.**CONTENT_ALPHA**
The surface will hold alpha content only.

cairocffi.**CONTENT_COLOR_ALPHA**
The surface will hold color and alpha content.

Pixel format

Used to identify the memory format of image data.

cairocffi.**FORMAT_ARGB32**
: Each pixel is a 32-bit quantity, with alpha in the upper 8 bits, then red, then green, then blue. The 32-bit quantities are stored native-endian. Pre-multiplied alpha is used. (That is, 50% transparent red is 0x80800000, not 0x80ff0000.)

cairocffi.**FORMAT_RGB24**
: Each pixel is a 32-bit quantity, with the upper 8 bits unused. Red, Green, and Blue are stored in the remaining 24 bits in that order.

cairocffi.**FORMAT_A8**
: Each pixel is a 8-bit quantity holding an alpha value.

cairocffi.**FORMAT_A1**
: Each pixel is a 1-bit quantity holding an alpha value. Pixels are packed together into 32-bit quantities. The ordering of the bits matches the endianess of the platform. On a big-endian machine, the first pixel is in the uppermost bit, on a little-endian machine the first pixel is in the least-significant bit.

cairocffi.**FORMAT_RGB16_565**
: Each pixel is a 16-bit quantity with red in the upper 5 bits, then green in the middle 6 bits, and blue in the lower 5 bits.

cairocffi.**FORMAT_RGB30**
: Like *FORMAT_RGB24*, but with the upper 2 bits unused and 10 bits per components.

Compositiong operator

Used to set the compositing operator for all cairo drawing operations.

The default operator is *OPERATOR_OVER*.

The operators marked as **unbounded** modify their destination even outside of the mask layer (that is, their effect is not bound by the mask layer). However, their effect can still be limited by way of clipping.

To keep things simple, the operator descriptions here document the behavior for when both source and destination are either fully transparent or fully opaque. The actual implementation works for translucent layers too. For a more detailed explanation of the effects of each operator, including the mathematical definitions, see http://cairographics.org/operators/.

cairocffi.**OPERATOR_CLEAR**
: Clear destination layer. (bounded)

cairocffi.**OPERATOR_SOURCE**
: Replace destination layer. (bounded)

cairocffi.**OPERATOR_OVER**
: Draw source layer on top of destination layer. (bounded)

cairocffi.**OPERATOR_IN**
: Draw source where there was destination content. (unbounded)

cairocffi.**OPERATOR_OUT**
: Draw source where there was no destination content. (unbounded)

cairocffi.**OPERATOR_ATOP**
: Draw source on top of destination content and only there.

cairocffi.**OPERATOR_DEST**
: Ignore the source.

cairocffi.**OPERATOR_DEST_OVER**
: Draw destination on top of source.

cairocffi.**OPERATOR_DEST_IN**
Leave destination only where there was source content. (unbounded)

cairocffi.**OPERATOR_DEST_OUT**
Leave destination only where there was no source content.

cairocffi.**OPERATOR_DEST_ATOP**
Leave destination on top of source content and only there. (unbounded)

cairocffi.**OPERATOR_XOR**
Source and destination are shown where there is only one of them.

cairocffi.**OPERATOR_ADD**
Source and destination layers are accumulated.

cairocffi.**OPERATOR_SATURATE**
Like *OPERATOR_OVER*, but assuming source and destination are disjoint geometries.

cairocffi.**OPERATOR_MULTIPLY**
Source and destination layers are multiplied. This causes the result to be at least as dark as the darker inputs. (Since 1.10)

cairocffi.**OPERATOR_SCREEN**
Source and destination are complemented and multiplied. This causes the result to be at least as light as the lighter inputs. (Since cairo 1.10)

cairocffi.**OPERATOR_OVERLAY**
Multiplies or screens, depending on the lightness of the destination color. (Since cairo 1.10)

cairocffi.**OPERATOR_DARKEN**
Replaces the destination with the source if it is darker, otherwise keeps the source. (Since cairo 1.10)

cairocffi.**OPERATOR_LIGHTEN**
Replaces the destination with the source if it is lighter, otherwise keeps the source. (Since cairo 1.10)

cairocffi.**OPERATOR_COLOR_DODGE**
Brightens the destination color to reflect the source color. (Since cairo 1.10)

cairocffi.**OPERATOR_COLOR_BURN**
Darkens the destination color to reflect the source color. (Since cairo 1.10)

cairocffi.**OPERATOR_HARD_LIGHT**
Multiplies or screens, dependent on source color. (Since cairo 1.10)

cairocffi.**OPERATOR_SOFT_LIGHT**
Darkens or lightens, dependent on source color. (Since cairo 1.10)

cairocffi.**OPERATOR_DIFFERENCE**
Takes the difference of the source and destination color. (Since cairo 1.10)

cairocffi.**OPERATOR_EXCLUSION**
Produces an effect similar to difference, but with lower contrast. (Since cairo 1.10)

cairocffi.**OPERATOR_HSL_HUE**
Creates a color with the hue of the source and the saturation and luminosity of the target. (Since cairo 1.10)

cairocffi.**OPERATOR_HSL_SATURATION**
Creates a color with the saturation of the source and the hue and luminosity of the target. Painting with this mode onto a gray area produces no change. (Since cairo 1.10)

cairocffi.**OPERATOR_HSL_COLOR**
Creates a color with the hue and saturation of the source and the luminosity of the target. This preserves the gray levels of the target and is useful for coloring monochrome images or tinting color images. (Since cairo 1.10)

cairocffi.**OPERATOR_HSL_LUMINOSITY**
Creates a color with the luminosity of the source and the hue and saturation of the target. This produces an inverse effect to *OPERATOR_HSL_COLOR*. (Since cairo 1.10)

Antialiasing mode

Specifies the type of antialiasing to do when rendering text or shapes.

cairocffi.**ANTIALIAS_DEFAULT**
Use the default antialiasing for the subsystem and target device.

cairocffi.**ANTIALIAS_NONE**
Use a bilevel alpha mask.

cairocffi.**ANTIALIAS_GRAY**
Perform single-color antialiasing.

cairocffi.**ANTIALIAS_SUBPIXEL**
Perform antialiasing by taking advantage of the order of subpixel elements on devices such as LCD panels.

As it is not necessarily clear from the above what advantages a particular antialias method provides, since cairo 1.12, there is also a set of hints:

cairocffi.**ANTIALIAS_FAST**
Allow the backend to degrade raster quality for speed.

cairocffi.**ANTIALIAS_GOOD**
A balance between speed and quality.

cairocffi.**ANTIALIAS_BEST**
A high-fidelity, but potentially slow, raster mode.

These make no guarantee on how the backend will perform its rasterisation (if it even rasterises!), nor that they have any differing effect other than to enable some form of antialiasing. In the case of glyph rendering, *ANTIALIAS_FAST* and *ANTIALIAS_GOOD* will be mapped to *ANTIALIAS_GRAY*, with *ANTIALIAS_BEST* being equivalent to *ANTIALIAS_SUBPIXEL*.

The interpretation of *ANTIALIAS_DEFAULT* is left entirely up to the backend, typically this will be similar to *ANTIALIAS_GOOD*.

Fill rule

Used to select how paths are filled. For both fill rules, whether or not a point is included in the fill is determined by taking a ray from that point to infinity and looking at intersections with the path. The ray can be in any direction, as long as it doesn't pass through the end point of a segment or have a tricky intersection such as intersecting tangent to the path. (Note that filling is not actually implemented in this way. This is just a description of the rule that is applied.)

The default fill rule is *FILL_RULE_WINDING*.

New entries may be added in future versions.

cairocffi.**FILL_RULE_WINDING**
If the path crosses the ray fromleft-to-right, counts +1. If the path crosses the rayfrom right to left, counts -1. (Left and right are determined from the perspective of looking along the ray from the starting point.) If the total count is non-zero, the point will be filled.

cairocffi.**FILL_RULE_EVEN_ODD**
Counts the total number of intersections, without regard to the orientation of the contour. If the total number of intersections is odd, the point will be filled.

Line cap style

Specifies how to render the endpoints of the path when stroking.

The default line cap style is *LINE_CAP_BUTT*.

cairocffi.**LINE_CAP_BUTT**
: Start (stop) the line exactly at the start (end) point.

cairocffi.**LINE_CAP_ROUND**
: Use a round ending, the center of the circle is the end point.

cairocffi.**LINE_CAP_SQUARE**
: Use squared ending, the center of the square is the end point.

Line join style

Specifies how to render the junction of two lines when stroking.

The default line join style is *LINE_JOIN_MITER*.

cairocffi.**LINE_JOIN_MITER**
: Use a sharp (angled) corner, see *Context.set_miter_limit()*.

cairocffi.**LINE_JOIN_ROUND**
: Use a rounded join, the center of the circle is the joint point.

cairocffi.**LINE_JOIN_BEVEL**
: Use a cut-off join, the join is cut off at half the line width from the joint point.

Font slant

Specifies variants of a font face based on their slant.

cairocffi.**FONT_SLANT_NORMAL**
: Upright font style.

cairocffi.**FONT_SLANT_ITALIC**
: Italic font style.

cairocffi.**FONT_SLANT_OBLIQUE**
: Oblique font style.

Font weight

Specifies variants of a font face based on their weight.

cairocffi.**FONT_WEIGHT_NORMAL**
: Normal font weight.

cairocffi.**FONT_WEIGHT_BOLD**
: Bold font weight.

Subpixel order

The subpixel order specifies the order of color elements within each pixel on the display device when rendering with an antialiasing mode of *ANTIALIAS_SUBPIXEL*.

cairocffi.**SUBPIXEL_ORDER_DEFAULT**
Use the default subpixel order for for the target device.

cairocffi.**SUBPIXEL_ORDER_RGB**
Subpixel elements are arranged horizontally with red at the left.

cairocffi.**SUBPIXEL_ORDER_BGR**
Subpixel elements are arranged horizontally with blue at the left.

cairocffi.**SUBPIXEL_ORDER_VRGB**
Subpixel elements are arranged vertically with red at the top.

cairocffi.**SUBPIXEL_ORDER_VBGR**
Subpixel elements are arranged vertically with blue at the top.

Hint style

Specifies the type of hinting to do on font outlines. Hinting is the process of fitting outlines to the pixel grid in order to improve the appearance of the result. Since hinting outlines involves distorting them, it also reduces the faithfulness to the original outline shapes. Not all of the outline hinting styles are supported by all font backends.

New entries may be added in future versions.

cairocffi.**HINT_STYLE_DEFAULT**
Use the default hint style for font backend and target device.

cairocffi.**HINT_STYLE_NONE**
Do not hint outlines.

cairocffi.**HINT_STYLE_SLIGHT**
Hint outlines slightly to improve contrast while retaining good fidelity to the original shapes.

cairocffi.**HINT_STYLE_MEDIUM**
Hint outlines with medium strength giving a compromise between fidelity to the original shapes and contrast.

cairocffi.**HINT_STYLE_FULL**
Hint outlines to maximize contrast.

Metrics hinting mode

Specifies whether to hint font metrics; hinting font metrics means quantizing them so that they are integer values in device space. Doing this improves the consistency of letter and line spacing, however it also means that text will be laid out differently at different zoom factors.

cairocffi.**HINT_METRICS_DEFAULT**
Hint metrics in the default manner for the font backend and target device.

cairocffi.**HINT_METRICS_OFF**
Do not hint font metrics.

cairocffi.**HINT_METRICS_ON**
Hint font metrics.

Path operation

Used to describe the type of one portion of a path when represented as a list. See *Context.copy_path()* for details.

cairocffi.**PATH_MOVE_TO**

cairocffi.**PATH_LINE_TO**

cairocffi.**PATH_CURVE_TO**

cairocffi.**PATH_CLOSE_PATH**

Pattern extend

Used to describe how pattern color/alpha will be determined for areas "outside" the pattern's natural area, (for example, outside the surface bounds or outside the gradient geometry).

Mesh patterns are not affected by the extend mode.

The default extend mode is *EXTEND_NONE* for *SurfacePattern* and *EXTEND_PAD* for *Gradient* patterns.

New entries may be added in future versions.

cairocffi.**EXTEND_NONE**
: Pixels outside of the source pattern are fully transparent.

cairocffi.**EXTEND_REPEAT**
: The pattern is tiled by repeating.

cairocffi.**EXTEND_REFLECT**
: The pattern is tiled by reflecting at the edges.

cairocffi.**EXTEND_PAD**
: Pixels outside of the pattern copy the closest pixel from the source.

Pixel filter

Used to indicate what filtering should be applied when reading pixel values from patterns. See *Pattern.set_filter()* for indicating the desired filter to be used with a particular pattern.

cairocffi.**FILTER_FAST**
: A high-performance filter, with quality similar to *FILTER_NEAREST*.

cairocffi.**FILTER_GOOD**
: A reasonable-performance filter, with quality similar to *FILTER_BILINEAR*.

cairocffi.**FILTER_BEST**
: The highest-quality available, performance may not be suitable for interactive use.

cairocffi.**FILTER_NEAREST**
: Nearest-neighbor filtering.

cairocffi.**FILTER_BILINEAR**
: Linear interpolation in two dimensions.

cairocffi.**FILTER_GAUSSIAN**
: This filter value is currently unimplemented, and should not be used in current code.

PDF version

Used to describe the version number of the PDF specification that a generated PDF file will conform to.

cairocffi.**PDF_VERSION_1_4**
: The version 1.4 of the PDF specification.

cairocffi.**PDF_VERSION_1_5**
The version 1.5 of the PDF specification.

PostScript level

Used to describe the language level of the PostScript Language Reference that a generated PostScript file will conform to.

cairocffi.**PS_LEVEL_2**
The language level 2 of the PostScript specification.

cairocffi.**PS_LEVEL_3**
The language level 3 of the PostScript specification.

SVG version

Used to describe the version number of the SVG specification that a generated SVG file will conform to.

cairocffi.**SVG_VERSION_1_1**
The version 1.1 of the SVG specification.

cairocffi.**SVG_VERSION_1_2**
The version 1.2 of the SVG specification.

Cluster flags

Specifies properties of a text cluster mapping. Flags are integer values representing a bit field.

cairocffi.**TEXT_CLUSTER_FLAG_BACKWARD = 0x00000001**
The clusters in the cluster array map to glyphs in the glyph array from end to start. (Since 1.8)

Decoding images with GDK-PixBuf

The *ImageSurface.create_from_png()* method can decode PNG images and provide a cairo surface, but what about other image formats?

The *cairocffi.pixbuf* module uses GDK-PixBuf to decode JPEG, GIF, and various other formats (depending on what is installed.) If you don't import this module, it is possible to use the rest of cairocffi without having GDK-PixBuf installed. GDK-PixBuf is an independent package since version 2.22, but before that was part of GTK+.

This module also converts pixel data since the internal format in GDK-PixBuf (big-endian RGBA) is not the same as in cairo (native-endian ARGB). For this reason, although it is a "toy" API, *ImageSurface.create_from_png()* can be faster than *decode_to_image_surface()* if the format is known to be PNG. The pixel conversion is done by GTK+ if available, but a (slower) fallback method is used otherwise.

exception cairocffi.pixbuf.**ImageLoadingError**
PixBuf returned an error when loading an image.

The image data is probably corrupted.

cairocffi.pixbuf.**decode_to_image_surface**(*image_data*, *width=None*, *height=None*)
Decode an image from memory into a cairo surface. The file format is detected automatically.

Parameters

- **image_data** – A byte string

- **width** – Integer width in pixels or None
- **height** – Integer height in pixels or None

Returns A tuple of a new *ImageSurface* object and the name of the detected image format.

Raises *ImageLoadingError* if the image data is invalid or in an unsupported format.

Using XCB surfaces with xcffib

The *cairocffi.xcb* module uses xcffib as the XCB library to create graphics for X windows and pixmaps.

CFFI API

cairocffi's *API* is made of a number of *wrapper* classes that provide a more Pythonic interface for various cairo objects. Functions that take a pointer as their first argument become methods, error statuses become exceptions, and *reference counting* is hidden away.

In order to use other C libraries that use integrate with cairo, or if cairocffi's API is not sufficient (Consider making a pull request!) you can access cairo's lower level C pointers and API through CFFI.

Module-level objects

`cairocffi.`**`ffi`**
A `cffi.FFI` instance with all of the cairo C API declared.

`cairocffi.`**`cairo`**
The libcairo library, pre-loaded with `ffi.dlopen()`. All cairo functions are accessible as attributes of this object:

```
import cairocffi
from cairocffi import cairo as cairo_c, SURFACE_TYPE_XLIB

if cairo_c.cairo_surface_get_type(surface._pointer) == SURFACE_TYPE_XLIB:
    ...
```

See the cairo manual for details.

Reference counting in cairo

Most cairo objects are reference-counted, and freed when the count reaches zero. cairocffi's Python wrapper will automatically decrease the reference count when they are garbage-collected. Therefore, care must be taken when creating a wrapper as to the reference count should be increased (for existing cairo objects) or not (for cairo objects that were just created with a refcount of 1.)

Wrappers

static `Surface.`**`_from_pointer`**(*pointer*, *incref*)
Wrap an existing `cairo_surface_t *` cdata pointer.

Parameters **incref** (*bool*) – Whether increase the *reference count* now.

Returns A new instance of *Surface* or one of its sub-classes, depending on the surface's type.

static Pattern.**_from_pointer**(*pointer*, *incref*)
Wrap an existing cairo_pattern_t * cdata pointer.

Parameters **incref** (*bool*) – Whether increase the *reference count* now.

Returns A new instance of *Pattern* or one of its sub-classes, depending on the pattern's type.

static FontFace.**_from_pointer**(*pointer*, *incref*)
Wrap an existing cairo_font_face_t * cdata pointer.

Parameters **incref** (*bool*) – Whether increase the *reference count* now.

Returns A new instance of *FontFace* or one of its sub-classes, depending on the face's type.

static ScaledFont.**_from_pointer**(*pointer*, *incref*)
Wrap an existing cairo_scaled_font_t * cdata pointer.

Parameters **incref** (*bool*) – Whether increase the *reference count* now.

Returns A new *ScaledFont* instance.

classmethod Context.**_from_pointer**(*pointer*, *incref*)
Wrap an existing cairo_t * cdata pointer.

Parameters **incref** (*bool*) – Whether increase the *reference count* now.

Returns A new *Context* instance.

Surface.**_pointer**
The underlying cairo_surface_t * cdata pointer.

Pattern.**_pointer**
The underlying cairo_pattern_t * cdata pointer.

FontFace.**_pointer**
The underlying cairo_font_face_t * cdata pointer.

ScaledFont.**_pointer**
The underlying cairo_scaled_font_t * cdata pointer.

FontOptions.**_pointer**
The underlying cairo_scaled_font_t * cdata pointer.

Matrix.**_pointer**
The underlying cairo_matrix_t * cdata pointer.

Context.**_pointer**
The underlying cairo_t * cdata pointer.

Converting pycairo wrappers to cairocffi

Some libraries such as PyGTK or PyGObject provide a pycairo Context object for you to draw on. It is possible to extract the underlying cairo_t * pointer and create a cairocffi wrapper for the same cairo context.

The follwing function does that with unsafe pointer manipulation. It only works on CPython.

```
# coding: utf-8

import cairo  # pycairo
import cairocffi
```

```
def _UNSAFE_pycairo_context_to_cairocffi(pycairo_context):
    # Sanity check. Continuing with another type would probably segfault.
    if not isinstance(pycairo_context, cairo.Context):
        raise TypeError('Expected a cairo.Context, got %r' % pycairo_context)

    # On CPython, id() gives the memory address of a Python object.
    # pycairo implements Context as a C struct:
    #     typedef struct {
    #         PyObject_HEAD
    #         cairo_t *ctx;
    #         PyObject *base;
    #     } PycairoContext;
    # Still on CPython, object.__basicsize__ is the size of PyObject_HEAD,
    # ie. the offset to the ctx field.
    # ffi.cast() converts the integer address to a cairo_t** pointer.
    # [0] dereferences that pointer, ie. read the ctx field.
    # The result is a cairo_t* pointer that cairocffi can use.
    return cairocffi.Context._from_pointer(
        cairocffi.ffi.cast('cairo_t **',
                           id(pycairo_context) + object.__basicsize__)[0],
        incref=True)
```

Converting other types of objects like surfaces is very similar, but left as an exercise to the reader.

Converting cairocffi wrappers to pycairo

The reverse conversion is also possible. Here we use ctypes rather than CFFI because Python's C API is sensitive to the GIL.

```
# coding: utf-8

import ctypes
import cairo  # pycairo
import cairocffi

pycairo = ctypes.PyDLL(cairo._cairo.__file__)
pycairo.PycairoContext_FromContext.restype = ctypes.c_void_p
pycairo.PycairoContext_FromContext.argtypes = 3 * [ctypes.c_void_p]
ctypes.pythonapi.PyList_Append.argtypes = 2 * [ctypes.c_void_p]


def _UNSAFE_cairocffi_context_to_pycairo(cairocffi_context):
    # Sanity check. Continuing with another type would probably segfault.
    if not isinstance(cairocffi_context, cairocffi.Context):
        raise TypeError('Expected a cairocffi.Context, got %r'
                        % cairocffi_context)

    # Create a reference for PycairoContext_FromContext to take ownership of.
    cairocffi.cairo.cairo_reference(cairocffi_context._pointer)
    # Casting the pointer to uintptr_t (the integer type as wide as a pointer)
    # gets the context's integer address.
    # On CPython id(cairo.Context) gives the address to the Context type,
    # as expected by PycairoContext_FromContext.
    address = pycairo.PycairoContext_FromContext(
        int(cairocffi.ffi.cast('uintptr_t', cairocffi_context._pointer)),
        id(cairo.Context),
```

```
        None)
    assert address
    # This trick uses Python's C API
    # to get a reference to a Python object from its address.
    temp_list = []
    assert ctypes.pythonapi.PyList_Append(id(temp_list), address) == 0
    return temp_list[0]
```

Example: using Pango through CFFI with cairocffi

The program below shows a fairly standard usage of CFFI to access Pango's C API. The *Context._pointer* pointer can be used directly as an argument to CFFI functions that expect `cairo_t *`. The C definitions are copied from Pango's and GLib's documentation.

Using CFFI for accessing Pango (rather than the traditional bindings in PyGTK or PyGObject with introspection) is not only easiest for using together with cairocffi, but also means that all of Pango's API is within reach, whereas bindings often only expose the high level API.

```
# coding: utf-8
import cairocffi
import cffi


ffi = cffi.FFI()
ffi.include(cairocffi.ffi)
ffi.cdef('''
    /* GLib */
    typedef void* gpointer;
    void g_object_unref (gpointer object);

    /* Pango and PangoCairo */
    typedef ... PangoLayout;
    typedef enum {
        PANGO_ALIGN_LEFT,
        PANGO_ALIGN_CENTER,
        PANGO_ALIGN_RIGHT
    } PangoAlignment;
    int pango_units_from_double (double d);
    PangoLayout * pango_cairo_create_layout (cairo_t *cr);
    void pango_cairo_show_layout (cairo_t *cr, PangoLayout *layout);
    void pango_layout_set_width (PangoLayout *layout, int width);
    void pango_layout_set_alignment (
        PangoLayout *layout, PangoAlignment alignment);
    void pango_layout_set_markup (
        PangoLayout *layout, const char *text, int length);
''')
gobject = ffi.dlopen('gobject-2.0')
pango = ffi.dlopen('pango-1.0')
pangocairo = ffi.dlopen('pangocairo-1.0')

gobject_ref = lambda pointer: ffi.gc(pointer, gobject.g_object_unref)
units_from_double = pango.pango_units_from_double


def write_example_pdf(target):
    pt_per_mm = 72 / 25.4
```

```
    width, height = 210 * pt_per_mm, 297 * pt_per_mm  # A4 portrait
    surface = cairocffi.PDFSurface(target, width, height)
    context = cairocffi.Context(surface)
    context.translate(0, 300)
    context.rotate(-0.2)

    layout = gobject_ref(
        pangocairo.pango_cairo_create_layout(context._pointer))
    pango.pango_layout_set_width(layout, units_from_double(width))
    pango.pango_layout_set_alignment(layout, pango.PANGO_ALIGN_CENTER)
    markup = u'<span font="italic 30">Hi from Παν!</span>'
    markup = ffi.new('char[]', markup.encode('utf8'))
    pango.pango_layout_set_markup(layout, markup, -1)
    pangocairo.pango_cairo_show_layout(context._pointer, layout)


if __name__ == '__main__':
    write_example_pdf(target='pango_example.pdf')
```

cairocffi changelog

Version 0.8.0

Released on 2017-02-03

- Follow semver
- #76: Avoid implicit relative import
- #74: Use utf-8 instead of utf8 in headers
- #73: Keep cairo library loaded until all relevant objects are freed
- #86: Add cairo_quartz_* functions for MacOS
- Use the default ReadTheDocs theme
- Fix implicit casts

Version 0.7.2

Released on 2015-08-04

- Use ctypes.util.find_library with dlopen.

Version 0.7.1

Released on 2015-06-22

- Allow installing cairocffi when cffi<1.0 is installed.

Version 0.7

Released on 2015-06-05

- #47: Fix PyPy support.
- #60: Use CFFI-1.0 methods.
- #61: Allow ffi import when package is pip installed.

Version 0.6

Released on 2014-09-23.

- #39: Add `xcb.XCBSurface`.
- #42: Add *`Win32PrintingSurface`*.

Version 0.5.4

Released on 2014-05-23.

- Stop testing with tox on Python 3.1, start on 3.4
- Start testing pushes and pull requests on Travis-CI
- Add more variants of the library names to try with *dlopen()*. This seems to be necessary on OpenBSD.

Version 0.5.3

Released on 2014-03-11.

Fix #28: Add another dynamic library name to try to load, for OS X.

Version 0.5.2

Released on 2014-02-27.

Fix #21: `UnicodeDecodeError` when installing with a non-UTF-8 locale.

Version 0.5.1

Released on 2013-07-16.

Fix #15: Work around CFFI bug #92 that caused memory leaks when file-like `target` objects are passed to *`Surface.write_to_png()`*, *`PDFSurface`*, *`PSSurface`* and *`SVGSurface`*.

Version 0.5

Released on 2013-06-20.

Change *`decode_to_image_surface()`* to raise a specific *`ImageLoadingError`* exception instead of a generic `ValueError`. This new exception type inherits from `ValueError`.

Version 0.4.3

Released on 2013-05-27.

- Fix #10: Pretend to be pycairo 1.10.0, for compatibility with matplotlib which does version detection.
- Fix WeasyPrint#94: Make (again??) GTK acutally optional for PixBuf support.

Version 0.4.2

Released on 2013-05-03.

- Fix #9: Make GTK acutally optional for PixBuf support.

Version 0.4.1

Released on 2013-04-30.

- Various documentation improvements
- Bug fixes:
  - Fix error handling in *ImageSurface.create_from_png()*.
  - Fix *ScaledFont.text_to_glyphs()* and *Context.show_text_glyphs()* with new-style enums.

Version 0.4

Released on 2013-04-06.

No change since 0.3.1, but depend on CFFI < 0.6 because of backward-incompatible changes. cairocffi 0.4 will require CFFI 0.6 or more.

```
# Before cairocffi 0.4:
surface = cairocffi.ImageSurface('ARGB32', 300, 400)

# All cairocffi versions:
surface = cairocffi.ImageSurface(cairocffi.FORMAT_ARGB32, 300, 400)
```

- Compatibility with CFFI 0.6

Version 0.3.2

Released on 2013-03-29.

No change since 0.3.1, but depend on CFFI < 0.6 because of backward-incompatible changes. cairocffi 0.4 will require CFFI 0.6 or more.

Version 0.3.1

Released on 2013-03-18.

Fix handling of GDK-PixBuf errors.

Version 0.3

Released on 2013-02-26.

- Add *cairocffi.pixbuf*, for loading images with GDK-PixBuf.
- Add iteration and item access on *Matrix*.
- Better Windows support by trying to load `libcairo-2.dll`

Version 0.2

Released on 2013-01-08.

Added *RecordingSurface*.

Version 0.1

Released on 2013-01-07.

First PyPI release.

Python Module Index

C

Index

Symbols

A

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y