pycairo Documentation

Release 1.8.10

Steve Chaplin

April 05, 2011

CONTENTS

CHAPTER

ONE

OVERVIEW

Pycairo is a Python binding for the cairo graphics library.

The Pycairo bindings are designed to match the cairo C API as closely as possible, and to deviate only in cases which are clearly better implemented in a more 'Pythonic' way.

Features of the Pycairo bindings:

- Provides an object oriented interface to cairo, using Python 2.2 new style classes.
- Pycairo_Check_Status() is called to check the status of cairo operations, and raise exceptions as appropriate.
- Provides a C API that can be used by other Python extensions.

The C cairo functions cairo_reference(), cairo_destroy(), cairo_surface_reference(), cairo_surface_destroy() (and their equivalents for surfaces and patterns) are not made public by the pycairo bindings. This is because pycairo handles cairo object construction and destruction.

To use the pycairo library:

```
import cairo
```

See *Reference* for further details.

For examples of pycairo code see the 'examples' directory that comes with the pycairo distribution.

CHAPTER

TWO

REFERENCE

2.1 Module Functions and Constants

2.1.1 Module Functions

cairo.**cairo_version**()

Returns the encoded version

Return type int

Returns the version of the underlying C cairo library, encoded in a single integer.

cairo.**cairo_version_string**()

Returns the encoded version

Return type str

Returns the version of the underlying C cairo library as a human-readable string of the form "X.Y.Z".

2.1.2 Module Constants

cairo.**version**
the pycairo version, as a string

cairo.**version_info**
the pycairo version, as a tuple

cairo.HAS

cairo.**HAS_ATSUI_FONT**
cairo.**HAS_FT_FONT**
cairo.**HAS_GLITZ_SURFACE**
cairo.**HAS_IMAGE_SURFACE**
cairo.**HAS_PDF_SURFACE**
cairo.**HAS_PNG_FUNCTIONS**

cairo.**HAS_PS_SURFACE**
cairo.**HAS_SVG_SURFACE**
cairo.**HAS_USER_FONT**
cairo.**HAS_QUARTZ_SURFACE**
cairo.**HAS_WIN32_FONT**
cairo.**HAS_WIN32_SURFACE**
cairo.**HAS_XCB_SURFACE**
cairo.**HAS_XLIB_SURFACE**
1 if the feature is present in the underlying C cairo library, 0 otherwise

cairo.ANTIALIAS

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

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

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

cairo.**ANTIALIAS_GRAY**
Perform single-color antialiasing (using shades of gray for black text on a white background, for example).

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

cairo.CONTENT

These constants are used to describe the content that a `Surface` will contain, whether color information, alpha information (translucence vs. opacity), or both.

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

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

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

cairo.EXTEND

These constants are 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).

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

cairo.**EXTEND_NONE**
pixels outside of the source pattern are fully transparent

cairo.**EXTEND_REPEAT**
the pattern is tiled by repeating

cairo.**EXTEND_REFLECT**
the pattern is tiled by reflecting at the edges (Implemented for surface patterns since 1.6)

cairo.**EXTEND_PAD**
pixels outside of the pattern copy the closest pixel from the source (Since 1.2; but only implemented for surface patterns since 1.6)

New entries may be added in future versions.

cairo.FILL_RULE

These constants are 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*.

cairo.**FILL_RULE_WINDING**
If the path crosses the ray from left-to-right, counts +1. If the path crosses the ray from 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.

cairo.**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.

New entries may be added in future versions.

cairo.FILTER

These constants are used to indicate what filtering should be applied when reading pixel values from patterns. See `SurfacePattern.set_filter()` for indicating the desired filter to be used with a particular pattern.

cairo.**FILTER_FAST**
A high-performance filter, with quality similar *FILTER_NEAREST*

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

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

cairo.**FILTER_NEAREST**
Nearest-neighbor filtering

cairo.**FILTER_BILINEAR**
Linear interpolation in two dimensions

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

cairo.FONT_SLANT

These constants specify variants of a FontFace based on their slant.

cairo.**FONT_SLANT_NORMAL**
Upright font style

cairo.**FONT_SLANT_ITALIC**
Italic font style

cairo.**FONT_SLANT_OBLIQUE**
Oblique font style

cairo.FONT_WEIGHT

These constants specify variants of a FontFace based on their weight.

cairo.**FONT_WEIGHT_NORMAL**
Normal font weight

cairo.**FONT_WEIGHT_BOLD**
Bold font weight

cairo.FORMAT

These constants are used to identify the memory format of ImageSurface data.

cairo.**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.)

cairo.**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.

cairo.**FORMAT_A8**
each pixel is a 8-bit quantity holding an alpha value.

cairo.**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.

New entries may be added in future versions.

cairo.HINT_METRICS

These constants specify 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.

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

`cairo.`**`HINT_METRICS_OFF`**
Do not hint font metrics

`cairo.`**`HINT_METRICS_ON`**
Hint font metrics

cairo.HINT_STYLE

These constants specify 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.

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

`cairo.`**`HINT_STYLE_NONE`**
Do not hint outlines

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

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

`cairo.`**`HINT_STYLE_FULL`**
Hint outlines to maximize contrast

New entries may be added in future versions.

cairo.LINE_CAP

These constants specify how to render the endpoints of the path when stroking.

The default line cap style is *LINE_CAP_BUTT*

cairo.**LINE_CAP_BUTT**
start(stop) the line exactly at the start(end) point

cairo.**LINE_CAP_ROUND**
use a round ending, the center of the circle is the end point

cairo.**LINE_CAP_SQUARE**
use squared ending, the center of the square is the end point

cairo.LINE_JOIN

These constants specify how to render the junction of two lines when stroking.

The default line join style is *LINE_JOIN_MITER*

cairo.**LINE_JOIN_MITER**
use a sharp (angled) corner, see `Context.set_miter_limit()`

cairo.**LINE_JOIN_ROUND**
use a rounded join, the center of the circle is the joint point

cairo.**LINE_JOIN_BEVEL**
use a cut-off join, the join is cut off at half the line width from the joint point

cairo.OPERATOR

These constants are 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.

cairo.**OPERATOR_CLEAR**
clear destination layer (bounded)

cairo.**OPERATOR_SOURCE**
replace destination layer (bounded)

cairo.**OPERATOR_OVER**
draw source layer on top of destination layer (bounded)

cairo.**OPERATOR_IN**
draw source where there was destination content (unbounded)

cairo.**OPERATOR_OUT**
draw source where there was no destination content (unbounded)

cairo.**OPERATOR_ATOP**
draw source on top of destination content and only there

cairo.**OPERATOR_DEST**
ignore the source

cairo.**OPERATOR_DEST_OVER**
draw destination on top of source

cairo.**OPERATOR_DEST_IN**
leave destination only where there was source content (unbounded)

cairo.**OPERATOR_DEST_OUT**
leave destination only where there was no source content

cairo.**OPERATOR_DEST_ATOP**
leave destination on top of source content and only there (unbounded)

cairo.**OPERATOR_XOR**
source and destination are shown where there is only one of them

cairo.**OPERATOR_ADD**
source and destination layers are accumulated

cairo.**OPERATOR_SATURATE**
like over, but assuming source and dest are disjoint geometries

cairo.PATH

These constants are used to describe the type of one portion of a path when represented as a Path.

cairo.**PATH_MOVE_TO**
A move-to operation

cairo.**PATH_LINE_TO**
A line-to operation

cairo.**PATH_CURVE_TO**
A curve-to operation

cairo.**PATH_CLOSE_PATH**
A close-path operation

cairo.PS_LEVEL

These constants are used to describe the language level of the PostScript Language Reference that a generated PostScript file will conform to. Note: the constants are only defined when cairo has been compiled with PS support enabled.

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

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

cairo.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.

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

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

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

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

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

2.2 Cairo Context

2.2.1 class Context()

Context is the main object used when drawing with cairo. To draw with cairo, you create a *Context*, set the target surface, and drawing options for the *Context*, create shapes with functions like Context.move_to() and Context.line_to(), and then draw shapes with Context.stroke() or Context.fill().

Contexts can be pushed to a stack via Context.save(). They may then safely be changed, without loosing the current state. Use Context.restore() to restore to the saved state.

class cairo.**Context**(*target*)

Parameters **target** – target Surface for the context

Returns a newly allocated *Context*

Raises *MemoryError* in case of no memory

Creates a new *Context* with all graphics state parameters set to default values and with *target* as a target surface. The target surface should be constructed with a backend-specific function such as ImageSurface (or any other cairo backend surface create variant).

append_path(*path*)

Parameters **path** – `Path` to be appended

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

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

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

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 `Context.new_sub_path()` before calling `Context.arc()`.

Angles are measured in radians. An angle of 0.0 is in the direction of the positive X axis (in user space). An angle of PI/2.0 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 * (math.pi / 180)`.

This function gives the arc in the direction of increasing angles; see `Context.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*:

```
ctx.save()
ctx.translate(x + width / 2., y + height / 2.)
ctx.scale(width / 2., height / 2.)
ctx.arc(0., 0., 1., 0., 2 * math.pi)
ctx.restore()
```

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

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

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 less than *angle1*.

See `Context.arc()` for more details. This function differs only in the direction of the arc between the two angles.

clip()

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

After `clip()`, the current path will be cleared from the `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 `Context.save()`/`Context.restore()` pair. The only other means of increasing the size of the clip region is `Context.reset_clip()`.

clip_extents()

Returns (x1, y1, x2, y2)

Return type (float, float, float, float)

- *x1*: left of the resulting extents
- *y1*: top of the resulting extents
- *x2*: right of the resulting extents
- *y2*: bottom of the resulting extents

Computes a bounding box in user coordinates covering the area inside the current clip. New in version 1.4.

clip_preserve()

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

Unlike `Context.clip()`, `clip_preserve()` preserves the path within the `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 `Context.save()`/`Context.restore()` pair. The only other means of increasing the size of the clip region is `Context.reset_clip()`.

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 `Context.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 `Context.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 function will have no effect.

Note: As of cairo version 1.2.4 any call to `close_path()` will place an explicit MOVE_TO element into the path immediately after the CLOSE_PATH element, (which can be seen in `Context.copy_path()` for example). This can simplify path processing in some cases as it may not be necessary to save the "last move_to point" during processing as the MOVE_TO immediately after the CLOSE_PATH will provide that point.

copy_clip_rectangle_list()

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

Return type list of 4-tuples of float

(The status in the list may be %CAIRO_STATUS_CLIP_NOT_REPRESENTABLE to indicate that the clip region cannot be represented as a list of user-space rectangles. The status may have other values to indicate other errors. - not implemented in pycairo) New in version 1.4.

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 `Context.show_page()` if you want to get an empty page after the emission.

This is a convenience function that simply calls `Surface.copy_page()` on *Context's* target.

copy_path()

Returns `Path`

Raises *MemoryError* in case of no memory

Creates a copy of the current path and returns it to the user as a `Path`.

copy_path_flat()

Returns `Path`

Raises *MemoryError* in case of no memory

Gets a flattened copy of the current path and returns it to the user as a `Path`.

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

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

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

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 function will behave as if preceded by a call to `ctx.move_to(x1, y1)`.

device_to_user(*x*, *y*)

Parameters

- **x** (*float*) – X value of coordinate
- **y** (*float*) – Y value of coordinate

Returns (x, y)

Return type (float, float)

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

device_to_user_distance(*dx*, *dy*)

Parameters

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

Returns (dx, dy)

Return type (float, float)

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

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 `Context`. See `Context.set_fill_rule()` and `Context.fill_preserve()`.

fill_extents()

Returns (x1, y1, x2, y2)

Return type (float, float, float, float)

- *x1*: left of the resulting extents
- *y1*: top of the resulting extents
- *x2*: right of the resulting extents
- *y2*: bottom of the resulting extents

Computes a bounding box in user coordinates covering the area that would be affected, (the "inked" area), by a `Context.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 `Context.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 `Context.path_extents()` may be more desirable for sake of performance if the non-inked path extents are desired.

See `Context.fill()`, `Context.set_fill_rule()` and `Context.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 `Context.fill()`, `fill_preserve()` preserves the path within the `Context`.

See `Context.set_fill_rule()` and `Context.fill()`.

font_extents()

Returns (ascent, descent, height, max_x_advance, max_y_advance)

Return type (float, float, float, float, float)

Gets the font extents for the currently selected font.

get_antialias()

Returns the current *ANTIALIAS* mode, as set by Context.set_antialias().

get_current_point()

Returns (x, y)

Return type (float, float)

- *x*: X coordinate of the current point
- *y*: Y coordinate of the current point

Gets 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 Context is in an error status, *x* and *y* will both be set to 0.0. It is possible to check this in advance with Context.has_current_point().

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

Some functions use and alter the current point but do not otherwise change current path: Context.show_text().

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

get_dash()

Returns (dashes, offset)

Return type (tuple, float)

- *dashes*: return value for the dash array
- *offset*: return value for the current dash offset

Gets the current dash array. New in version 1.4.

get_dash_count()

Returns the length of the dash array, or 0 if no dash array set.

Return type int

See also `Context.set_dash()` and `Context.get_dash()`. New in version 1.4.

get_fill_rule()

Returns the current *FILL RULE*, as set by `Context.set_fill_rule()`.

get_font_face()

Returns the current `FontFace` for the `Context`.

get_font_matrix()

Returns the current `Matrix` for the `Context`.

See `Context.set_font_matrix()`.

get_font_options()

Returns the current `FontOptions` for the `Context`.

Retrieves font rendering options set via `Context.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 `Context.set_font_options()`.

get_group_target()

Returns the target `Surface`.

Gets 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 `Context.push_group()` or `Context.push_group_with_content()`. New in version 1.2.

get_line_cap()

Returns the current *LINE_CAP* style, as set by `Context.set_line_cap()`.

get_line_join()

Returns the current *LINE_JOIN* style, as set by `Context.set_line_join()`.

get_line_width()

Returns the current line width

Return type float

This function returns the current line width value exactly as set by `Context.set_line_width()`. Note that the value is unchanged even if the CTM has changed between the calls to `Context.set_line_width()` and `get_line_width()`.

get_matrix()

Returns the current transformation Matrix (CTM)

get_miter_limit()

Returns the current miter limit, as set by Context.set_miter_limit().

Return type float

get_operator()

Returns the current compositing *OPERATOR* for a Context.

get_scaled_font()

Returns the current ScaledFont for a Context.

New in version 1.4.

get_source()

Returns the current source Pattern for a Context.

get_target()

Returns the target Surface for the Context

get_tolerance()

Returns the current tolerance value, as set by Context.set_tolerance()

Return type float

glyph_extents(*glyphs*[, *num_glyphs*])

Parameters

- **glyphs** (*a sequence of (int, float, float)*) – glyphs
- **num_glyphs** (*int*) – number of glyphs to measure, defaults to using all

Returns x_bearing, y_bearing, width, height, x_advance, y_advance

Return type 6-tuple of float

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

Note that whitespace glyphs do not contribute to the size of the rectangle (extents.width and extents.height).

glyph_path(*glyphs*[, *num_glyphs*])

Parameters

- **glyphs** (*a sequence of (int, float, float)*) – glyphs to show
- **num_glyphs** (*int*) – number of glyphs to show, defaults to showing all

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

has_current_point()

returns: True iff a current point is defined on the current path. See `Context.get_current_point()` for details on the current point.

New in version 1.6.

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.

in_fill(*x*, *y*)

Parameters

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

Returns True iff the point is inside the area that would be affected by a `Context.fill()` operation given the current path and filling parameters. Surface dimensions and clipping are not taken into account.

See `Context.fill()`, `Context.set_fill_rule()` and `Context.fill_preserve()`.

in_stroke(*x*, *y*)

Parameters

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

Returns True iff the point is inside the area that would be affected by a `Context.stroke()` operation given the current path and stroking parameters. Surface dimensions and clipping are not taken into account.

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

line_to(*x*, *y*)

Parameters

- **x** (*float*) – the X coordinate of the end of the new line
- **y** (*float*) – the Y coordinate of the end of the new line

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 function will behave as `ctx.move_to(x, y)`.

mask (*pattern*)

Parameters **pattern** – a `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.)

mask_surface (*surface*, *x=0.0*, *y=0.0*)

Parameters

- **surface** – a `Surface`
- **x** (*float*) – X coordinate at which to place the origin of *surface*
- **y** (*float*) – Y coordinate at which to place the origin of *surface*

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.)

move_to (*x*, *y*)

Parameters

- **x** (*float*) – the X coordinate of the new position
- **y** (*float*) – the Y coordinate of the new position

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

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 `Context.move_to()`.

A call to `new_sub_path()` is particularly useful when beginning a new sub-path with one of the `Context.arc()` calls. This makes things easier as it is no longer necessary to manually compute the arc's initial coordinates for a call to `Context.move_to()`. New in version 1.6.

paint ()

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

paint_with_alpha (*alpha*)

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

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 `Context.paint()`, but the drawing is faded out using the alpha value.

path_extents()

Returns (x1, y1, x2, y2)

Return type (float, float, float, float)

- *x1*: left of the resulting extents
- *y1*: top of the resulting extents
- *x2*: right of the resulting extents
- *y2*: bottom of the resulting 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 `Context.fill_extents()` and `Context.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 `Context.stroke_extents()` with cairo.LINE_CAP_ROUND as the line width approaches 0.0, (but never reaching the empty-rectangle returned by `Context.stroke_extents()` for a line width of 0.0).

Specifically, this means that zero-area sub-paths such as `Context.move_to()`; `Context.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 `Context.move_to()` will not contribute to the results of `Context.path_extents()`. New in version 1.6.

pop_group()

Returns a newly created `SurfacePattern` containing the results of all drawing operations performed to the group.

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

The `pop_group()` function calls `Context.restore()`, (balancing a call to `Context.save()` by the `Context.push_group()` function), so that any changes to the graphics state will not be visible outside the group. New in version 1.2.

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

The behavior of this function is equivalent to the sequence of operations:

```
group = cairo_pop_group()
ctx.set_source(group)
```

but is more convenient as their is no need for a variable to store the short-lived pointer to the pattern.

The Context.pop_group() function calls Context.restore(), (balancing a call to Context.save() by the Context.push_group() function), so that any changes to the graphics state will not be visible outside the group. New in version 1.2.

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 Context.pop_group() or Context.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 Context.push_group()/Context.pop_group(). Each call pushes/pops the new target group onto/from a stack.

The push_group() function calls Context.save() so that any changes to the graphics state will not be visible outside the group, (the pop_group functions call Context.restore()).

By default the intermediate group will have a *CONTENT* type of cairo.CONTENT_COLOR_ALPHA. Other content types can be chosen for the group by using Context.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:

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

New in version 1.2.

push_group_with_content(*content*)

Parameters **content** – a *CONTENT* indicating the type of group that will be created

Temporarily redirects drawing to an intermediate surface known as a group. The redirection lasts until the group is completed by a call to `Context.pop_group()` or `Context.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 function and `Context.push_group()` which you should see for a more detailed description of group rendering. New in version 1.2.

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

Parameters

- **x** (*float*) – the X coordinate of the top left corner of the rectangle
- **y** (*float*) – the Y coordinate to the top left corner of the rectangle
- **width** (*float*) – the width of the rectangle
- **height** (*float*) – the height of the rectangle

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

This function is logically equivalent to:

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

rel_curve_to(*dx1*, *dy1*, *dx2*, *dy2*, *dx3*, *dy4*)

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 `cairo.Error` if called with no current point.

Relative-coordinate version of `Context.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), `ctx.rel_curve_to(dx1, dy1, dx2, dy2, dx3, dy3)` is logically equivalent to `ctx.curve_to(x+dx1, y+dy1, x+dx2, y+dy2, x+dx3, y+dy3)`.

rel_line_to(*dx*, *dy*)

Parameters

- **dx** (*float*) – the X offset to the end of the new line
- **dy** (*float*) – the Y offset to the end of the new line

Raises `cairo.Error` if called with no current point.

Relative-coordinate version of `Context.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), `ctx.rel_line_to(dx, dy)` is logically equivalent to `ctx.line_to(x + dx, y + dy)`.

rel_move_to(*dx*, *dy*)

Parameters

- **dx** (*float*) – the X offset
- **dy** (*float*) – the Y offset

Raises `cairo.Error` if called with no current point.

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

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

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 `save()` and `restore()` around `clip()` as a more robust means of temporarily restricting the clip region.

restore()

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

rotate(*angle*)

Parameters angle (*float*) – angle (in radians) by which the user-space axes will be rotated

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. The rotation direction for positive angles is from the positive X axis toward the positive Y axis.

save()

Makes a copy of the current state of `Context` and saves it on an internal stack of saved states. When `restore()` is called, `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()`.

scale(*sx*, *sy*)

Parameters

- **sx** (*float*) – scale factor for the X dimension
- **sy** (*float*) – scale factor for the Y dimension

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.

select_font_face(*family*[, *slant*[, *weight*]])

Parameters

- **family** (*str or unicode*) – a font family name
- **slant** – the *FONT_SLANT* of the font, defaults to `cairo.FONT_SLANT_NORMAL`.
- **weight** – the *FONT_WEIGHT* of the font, defaults to `cairo.FONT_WEIGHT_NORMAL`.

Note: The `select_font_face()` function call 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.

Selects a family and style of font from a simplified description as a family name, slant and weight. 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.

For "real" font selection, see the font-backend-specific font_face_create functions for the font backend you are using. (For example, if you are using the freetype-based cairo-ft font backend, see cairo_ft_font_face_create_for_ft_face() or cairo_ft_font_face_create_for_pattern().) The resulting font face could then be used with cairo_scaled_font_create() and cairo_set_scaled_font().

Similarly, when using the "real" font support, you can call directly into the underlying font system, (such as fontconfig or freetype), for operations such as listing

available fonts, etc.

It is expected that most applications will need to use a more comprehensive font handling and text layout library, (for example, pango), in conjunction with cairo.

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 cairo.FONT_SLANT_NORMAL, and default weight is cairo.FONT_WEIGHT_NORMAL.

This function is equivalent to a call to `ToyFontFace` followed by `set_font_face()`.

set_antialias (*antialias*)

Parameters **antialias** – the new *ANTIALIAS* mode

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 `cairo.ANTIALIAS_SUBPIXEL` when drawing shapes.

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

set_dash (*dashes*[, *offset=0*])

Parameters

- **dashes** (*sequence of float*) – a sequence specifying alternate lengths of on and off stroke portions.
- **offset** (*int*) – an offset into the dash pattern at which the stroke should start, defaults to 0.

Raises `cairo.Error` if any value in *dashes* is negative, or if all values are 0.

Sets the dash pattern to be used by `stroke()`. A dash pattern is specified by *dashes* - a sequence of positive values. Each value provides the length of alternate "on" and "off" portions of the stroke. The *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 0.0 with `cairo.LINE_CAP_ROUND` or `cairo.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 the number of dashes is 0 dashing is disabled.

If the number of dashes is 1 a symmetric pattern is assumed with alternating on and off portions of the size specified by the single value in *dashes*.

set_fill_rule(*fill_rule*)

Parameters **fill_rule** – a *FILL RULE* to set the 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 `cairo.FILL_RULE_WINDING`.

set_font_face(*font_face*)

Parameters **font_face** – a `FontFace`, or None to restore to the default `FontFace`

Replaces the current `FontFace` object in the `Context` with *font_face*.

set_font_matrix(*matrix*)

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

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

set_font_options(*options*)

Parameters **options** – `FontOptions` to use

Sets a set of custom font rendering options for the `Context`. 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 `cairo.ANTIALIAS_DEFAULT`), then the value from the surface is used.

set_font_size(*size*)

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

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.

set_line_cap(*line_cap*)

Parameters **line_cap** – a *LINE_CAP* style

Sets the current line cap style within the `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 style is `cairo.LINE_CAP_BUTT`.

set_line_join(*line_join*)

Parameters **line_join** – a *LINE_JOIN* style

Sets the current line join style within the `Context`.

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

The default line join style is `cairo.LINE_JOIN_MITER`.

set_line_width(*width*)

Parameters **width** (*float*) – a line width

Sets the current line width within the `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 width 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.

set_matrix(*matrix*)

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

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

set_miter_limit(*limit*)

Parameters **limit** – miter limit to set

Sets the current miter limit within the `Context`.

If the current line join style is set to `cairo.LINE_JOIN_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 miter limit 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/math.sin(angle/2)
```

set_operator(*op*)

Parameters **op** – the compositing *OPERATOR* to set for use in all drawing operations.

The default operator is `cairo.OPERATOR_OVER`.

set_scaled_font(*scaled_font*)

Parameters **scaled_font** – a `ScaledFont`

Replaces the current font face, font matrix, and font options in the `Context` with those of the `ScaledFont`. Except for some translation, the current CTM of the `Context` should be the same as that of the `ScaledFont`, which can be accessed using `ScaledFont.get_ctm()`. New in version 1.2.

set_source(*source*)

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

Sets the source pattern within `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 a solid pattern that is opaque black, (that is, it is equivalent to `set_source_rgb(0.0, 0.0, 0.0)`.

set_source_rgb(*red*, *green*, *blue*)

Parameters

- **red** (*float*) – red component of color
- **green** (*float*) – green component of color
- **blue** (*float*) – blue component of color

Sets the source pattern within `Context` to an opaque color. This opaque color will then be used for any subsequent drawing operation until a new source pattern is set.

The color 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 `set_source_rgb(0.0, 0.0, 0.0)`.

set_source_rgba(*red*, *green*, *blue*[, *alpha=1.0*])

Parameters

- **red** (*float*) – red component of color
- **green** (*float*) – green component of color
- **blue** (*float*) – blue component of color
- **alpha** (*float*) – alpha component of color

Sets the source pattern within `Context` to a translucent 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 `set_source_rgba(0.0, 0.0, 0.0, 1.0)`.

set_source_surface(*surface*[, *x=0.0*[, *y=0.0*]])

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

This is a convenience function for creating a pattern from a `Surface` and setting it as the source in `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* patterns 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()`).

set_tolerance(*tolerance*)

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

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.

show_glyphs (*glyphs*[, *num_glyphs*])

Parameters

- **glyphs** (*a sequence of (int, float, float)*) – glyphs to show
- **num_glyphs** (*int*) – number of glyphs to show, defaults to showing all glyphs

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

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 function that simply calls `ctx.get_target() . show_page()`

show_text (*text*)

Parameters **text** (*str or unicode*) – text

A drawing operator that generates the shape from a string of text, rendered according to the current font_face, font_size (font_matrix), and font_options.

This function 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()`.

Note: The `show_text()` function call 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 `show_glyphs()` for the "real" text display API in cairo.

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 `cairo.LINE_CAP_ROUND` or `cairo.LINE_CAP_SQUARE` then these segments will be drawn as circular dots or squares respectively. In the case of `cairo.LINE_CAP_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 `cairo.LINE_CAP_ROUND` then these sub-paths will be drawn as circular dots. Note that in the case of `cairo.LINE_CAP_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 `cairo.LINE_CAP_BUTT` cause anything to be drawn in the case of either degenerate segments or sub-paths.

stroke_extents()

Returns (x1, y1, x2, y2)

Return type (float, float, float, float)

- *x1*: left of the resulting extents
- *y1*: top of the resulting extents
- *x2*: right of the resulting extents
- *y2*: bottom of the resulting extents

Computes a bounding box in user 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 non-inked path extents are desired.

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

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_preserve()`.

text_extents(*text*)

Parameters **text** (*string or unicode*) – text to get extents for

Returns x_bearing, y_bearing, width, height, x_advance, y_advance

Return type 6-tuple of float

Gets 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 `Context.show_text()`). Additionally, the x_advance and y_advance values indicate the amount by which the current point would be advanced by `Context.show_text()`.

Note that whitespace characters do not directly contribute to the size of the rectangle (extents.width and extents.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.

text_path(*text*)

Parameters **text** (*string or unicode*) – text

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

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

Like `Context.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 `Context.text_path()` without having to set current point in between.

Note: The `text_path()` function call 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 `Context.glyph_path()` for the "real" text path API in cairo.

transform(*matrix*)

Parameters **matrix** – a transformation `Matrix` to be applied to the user-space axes

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.

translate(*tx*, *ty*)

Parameters

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

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.

user_to_device (*x*, *y*)

Parameters

- **x** (*float*) – X value of coordinate
- **y** (*float*) – Y value of coordinate

Returns (x, y)

Return type (float, float)

•*x*: X value of coordinate

•*y*: Y value of coordinate

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

user_to_device_distance (*dx*, *dy*)

Parameters

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

Returns (dx, dy)

Return type (float, float)

•*dx*: X value of a distance vector

•*dy*: Y value of a distance vector

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

2.3 Exceptions

When a cairo function or method call fails an exception is raised. I/O errors raise IOError, memory errors raise MemoryError, and all other errors raise cairo.Error.

2.3.1 cairo.Error()

exception `cairo.`**Error**

This exception is raised when a cairo object returns an error status.

2.4 Matrix

2.4.1 class Matrix()

Matrix is 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.

Some standard Python operators can be used with matrices:

To read the values from a *Matrix*:

```
xx, yx, xy, yy, x0, y0 = matrix
```

To multiply two matrices:

```
matrix3 = matrix1.multiply(matrix2)
# or equivalently
matrix3 = matrix1 * matrix2
```

To compare two matrices:

```
matrix1 == matrix2
matrix1 != matrix2
```

For more information on matrix transformation see http://www.cairographics.org/matrix_transform

class `cairo.`**`Matrix`**(*xx = 1.0*, *yx = 0.0*, *xy = 0.0*, *yy = 1.0*, *x0 = 0.0*, *y0 = 0.0*)

Parameters

- **xx** (*float*) – xx component of the affine transformation
- **yx** (*float*) – yx component of the affine transformation
- **xy** (*float*) – xy component of the affine transformation
- **yy** (*float*) – yy component of the affine transformation
- **x0** (*float*) – X translation component of the affine transformation
- **y0** (*float*) – Y translation component of the affine transformation

Create a new *Matrix* with the affine transformation given by *xx, yx, xy, yy, x0, y0*. The transformation is given by:

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

To create a new identity matrix:

```
matrix = cairo.Matrix()
```

To create a matrix with a transformation which translates by tx and ty in the X and Y dimensions, respectively:

```
matrix = cairo.Matrix(x0=tx, y0=ty)
```

To create a matrix with a transformation that scales by sx and sy in the X and Y dimensions, respectively:

```
matrix = cairo.Matrix(xx=sy, yy=sy)
```

classmethod **init_rotate**(*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.

Returns a new *Matrix* set to a transformation that rotates by *radians*.

invert()

Returns If *Matrix* has an inverse, modifies *Matrix* to be the inverse matrix and returns *None*

Raises `cairo.Error` if the *Matrix* as no inverse

Changes *Matrix* to be the inverse of it's 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.

multiply(*matrix2*)

Parameters **matrix2** (*cairo.Matrix*) – a second matrix

Returns a new *Matrix*

Multiplies the affine transformations in *Matrix* and *matrix2* together. The effect of the resulting transformation is to first apply the transformation in *Matrix* to the coordinates and then apply the transformation in *matrix2* to the coordinates.

It is allowable for result to be identical to either *Matrix* or *matrix2*.

rotate(*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.

Initialize *Matrix* to a transformation that rotates by *radians*.

scale(*sx*, *sy*)

Parameters

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

Applies scaling by *sx, sy* to the transformation in *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.

transform_distance(*dx*, *dy*)

Parameters

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

Returns the transformed distance vector (dx,dy)

Return type (float, float)

Transforms the distance vector *(dx,dy)* by *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 * a + dy1 * c
dy2 = dx1 * b + dy1 * d
```

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*.

transform_point(*x*, *y*)

Parameters

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

Returns the transformed point (x,y)

Return type (float, float)

Transforms the point *(x, y)* by *Matrix*.

translate(*tx*, *ty*)

Parameters

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

Applies a transformation by *tx, ty* to the transformation in *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.

2.5 Paths

2.5.1 class Path()

class `cairo.`**`Path`**

Path cannot be instantiated directly, it is created by calling `Context.copy_path()` and `Context.copy_path_flat()`.

str(path) lists the path elements.

See *PATH attributes*

Path is an iterator.

See examples/warpedtext.py for example usage.

2.6 Patterns

Patterns are the paint with which cairo draws. The primary use of patterns is as the source for all cairo drawing operations, although they can also be used as masks, that is, as the brush too.

A cairo *Pattern* is created by using one of the *PatternType* constructors listed below, or implicitly through *Context.set_source_<type>()* methods.

2.6.1 class Pattern()

Pattern is the abstract base class from which all the other pattern classes derive. It cannot be instantiated directly.

class `cairo.`**`Pattern`**

`get_extend``()`

Returns the current extend strategy used for drawing the *Pattern*.

Return type int

Gets the current extend mode for the *Pattern*. See *EXTEND attributes* for details on the semantics of each extend strategy.

`get_matrix``()`

Returns a new `Matrix` which stores a copy of the *Pattern's* transformation matrix

`set_extend``(`*extend*`)`

Parameters **extend** – an *EXTEND* describing how the area outside of the *Pattern* will be drawn

Sets the mode to be used for drawing outside the area of a *Pattern*.

The default extend mode is `cairo.EXTEND_NONE` for `SurfacePattern` and `cairo.EXTEND_PAD` for `Gradient` Patterns.

set_matrix(*matrix*)

Parameters **matrix** – a `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:

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

Meanwhile, using values of 2.0 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`.

2.6.2 class SolidPattern(Pattern)

class cairo.**SolidPattern**(*red*, *green*, *blue*, *alpha=1.0*)

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

Returns a new *SolidPattern*

Raises *MemoryError* in case of no memory

Creates a new *SolidPattern* corresponding to a translucent color. The color components are floating point numbers in the range 0 to 1. If the values passed in are outside that range, they will be clamped.

get_rgba()

Returns (red, green, blue, alpha) a tuple of float

Gets the solid color for a *SolidPattern*. New in version 1.4.

2.6.3 class SurfacePattern(Pattern)

class cairo.**SurfacePattern**(*surface*)

Parameters **surface** – a cairo Surface

Returns a newly created *SurfacePattern* for the given surface.

Raises *MemoryError* in case of no memory.

get_filter()

Returns the current *FILTER* used for resizing the *SurfacePattern*.

get_surface()

Returns the Surface of the *SurfacePattern*.

New in version 1.4.

set_filter(*filter*)

Parameters **filter** – a *FILTER* describing the filter to use for resizing the *Pattern*

Note that you might want to control filtering even when you do not have an explicit *Pattern* object, (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.set_source_surface(image, x, y)
surfacepattern.set_filter(context.get_source(), cairo.FILTER_NEAREST)
```

2.6.4 class Gradient(Pattern)

Gradient is an abstract base class from which other *Pattern* classes derive. It cannot be instantiated directly.

class cairo.**Gradient**

add_color_stop_rgb(*offset*, *red*, *green*, *blue*)

Parameters

- **offset** (*float*) – an offset in the range [0.0 .. 1.0]
- **red** (*float*) – red component of color
- **green** (*float*) – green component of color
- **blue** (*float*) – blue component of color

Adds an opaque color stop to a *Gradient* pattern. The offset specifies the location along the gradient's control vector. For example, a *LinearGradient's* control vector is from (x0,y0) to (x1,y1) while a *RadialGradient's* control vector is from any point on the start circle to the corresponding point on the end circle.

The color is specified in the same way as in `Context.set_source_rgb()`.

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 will compare less than stops added later). This can be useful for reliably making sharp color transitions instead of the typical blend.

add_color_stop_rgba (*offset*, *red*, *green*, *blue*, *alpha*)

Parameters

- **offset** (*float*) – an offset in the range [0.0 .. 1.0]
- **red** (*float*) – red component of color
- **green** (*float*) – green component of color
- **blue** (*float*) – blue component of color
- **alpha** (*float*) – alpha component of color

Adds an opaque color stop to a *Gradient* pattern. The offset specifies the location along the gradient's control vector. For example, a *LinearGradient's* control vector is from (x0,y0) to (x1,y1) while a *RadialGradient's* control vector is from any point on the start circle to the corresponding point on the end circle.

The color is specified in the same way as in `Context.set_source_rgb()`.

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 will compare less than stops added later). This can be useful for reliably making sharp color transitions instead of the typical blend.

2.6.5 class LinearGradient(`Gradient`)

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

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

Returns a new *LinearGradient*

Raises *MemoryError* in case of no memory

Create a new *LinearGradient* along the line defined by (x0, y0) and (x1, y1). Before using the *Gradient* pattern, a number of color stops should be defined using `Gradient.add_color_stop_rgb()` or `Gradient.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 `Pattern.set_matrix()`

get_linear_points()

Returns

(x0, y0, x1, y1) - a tuple of float

- x0: return value for the x coordinate of the first point
- y0: return value for the y coordinate of the first point
- x1: return value for the x coordinate of the second point
- y1: return value for the y coordinate of the second point

Gets the gradient endpoints for a *LinearGradient*. New in version 1.4.

2.6.6 class RadialGradient(`Gradient`)

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

Parameters

- **cx0** (*float*) – x coordinate for the center of the start circle
- **cy0** (*float*) – y coordinate for the center of the start circle
- **radius0** (*float*) – radius of the start circle
- **cx1** (*float*) – x coordinate for the center of the end circle
- **cy1** (*float*) – y coordinate for the center of the end circle
- **radius1** (*float*) – radius of the end circle

Returns the newly created *RadialGradient*

Raises *MemoryError* in case of no memory

Creates a new *RadialGradient* 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 `Gradient.add_color_stop_rgb()` or `Gradient.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 `Pattern.set_matrix()`.

get_radial_circles()

Returns

(x0, y0, r0, x1, y1, r1) - a tuple of float

- x0: return value for the x coordinate of the center of the first circle
- y0: return value for the y coordinate of the center of the first circle
- r0: return value for the radius of the first circle
- x1: return value for the x coordinate of the center of the second circle
- y1: return value for the y coordinate of the center of the second circle
- r1: return value for the radius of the second circle

Gets the *Gradient* endpoint circles for a *RadialGradient*, each specified as a center coordinate and a radius. New in version 1.4.

2.7 Surfaces

cairo.Surface is the abstract type representing all different drawing targets that cairo can render to. The actual drawings are performed using a `Context`.

A cairo.Surface is created by using backend-specific constructors of the form cairo.<XXX>Surface().

2.7.1 class Surface()

class `cairo.`**`Surface`**

Surface is the abstract base class from which all the other surface classes derive. It cannot be instantiated directly.

`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.

`Context.copy_page()` is a convenience function for this. New in version 1.6.

`create_similar`(*content*, *width*, *height*)

Parameters

- **content** – the *CONTENT* for the new surface
- **width** (*int*) – width of the new surface, (in device-space units)
- **height** – height of the new surface (in device-space units)

Returns a newly allocated *Surface*.

Create a *Surface* that is as compatible as possible with the existing surface. For example the new surface will have the same fallback resolution and `FontOptions`. Generally, the new surface will also use the same backend, unless that is not possible for some reason.

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

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 flushing and finishing it. Further drawing to the surface will not affect the surface but will instead trigger a `cairo.Error` exception.

flush()

Do any pending drawing for the *Surface* and also restore any temporary modification's 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 function does nothing.

get_content()

Returns The *CONTENT* type of *Surface*, which indicates whether the *Surface* contains color and/or alpha information.

New in version 1.2.

get_device_offset()

Returns

(x_offset, y_offset) a tuple of float

- x_offset: the offset in the X direction, in device units
- y_offset: the offset in the Y direction, in device units

This method returns the previous device offset set by `set_device_offset()`. New in version 1.2.

get_fallback_resolution()

Returns

(x_pixels_per_inch, y_pixels_per_inch) a tuple of float

- x_pixels_per_inch: horizontal pixels per inch
- y_pixels_per_inch: vertical pixels per inch

This method returns the previous fallback resolution set by `set_fallback_resolution()`, or default fallback resolution if never set. New in version 1.8.

get_font_options()

Returns a `FontOptions`

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`.

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*)

Parameters

- **x** (*int*) – X coordinate of dirty rectangle
- **y** (*int*) – Y coordinate of dirty rectangle
- **width** (*int*) – width of dirty rectangle
- **height** (*int*) – height of dirty rectangle

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 function, to make sure that future cairo calls have the clip set that they expect.

set_device_offset(*x_offset*, *y_offset*)

Parameters

- **x_offset** (*float*) – the offset in the X direction, in device units
- **y_offset** (*float*) – the offset in the Y direction, in device units

Sets an offset that is added to the device coordinates determined by the CTM when drawing to *Surface*. One use case for this function 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 functions 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.

set_fallback_resolution(*x_pixels_per_inch*, *y_pixels_per_inch*)

Parameters

- **x_pixels_per_inch** (*float*) – horizontal setting for pixels per inch
- **y_pixels_per_inch** (*float*) – vertical setting for 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 function 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 function has no effect on those backends.

Note: The fallback resolution only takes effect at the time of completing a page (with `Context.show_page()` or `Context.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. New in version 1.2.

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.

There is a convenience function for this that takes a `Context.show_page()`. New in version 1.6.

write_to_png(*fobj*)

Parameters **fobj** (*filename (str or unicode), file or file-like object*) – the file to write to

Raises *MemoryError* if memory could not be allocated for the operation

IOError if an I/O error occurs while attempting to write the file

Writes the contents of *Surface* to *fobj* as a PNG image.

2.7.2 class ImageSurface(Surface)

A *cairo.ImageSurface* provides the ability to render to memory buffers either allocated by cairo or by the calling code. The supported image formats are those defined in *FORMAT attributes*.

class cairo.**ImageSurface**(*format*, *width*, *height*)

Parameters

- **format** – *FORMAT* of pixels in the surface to create
- **width** – width of the surface, in pixels
- **height** – height of the surface, in pixels

Returns a new *ImageSurface*

Raises *MemoryError* in case of no memory

Creates an *ImageSurface* of the specified format and dimensions. Initially the surface contents are all 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).

classmethod **`create_for_data`** (*data*, *format*, *width*, *height*[, *stride*])

Parameters

- **data** – a writable Python buffer object
- **format** – the *FORMAT* of pixels in the buffer
- **width** – the width of the image to be stored in the buffer
- **height** – the height of the image to be stored in the buffer
- **stride** – the number of bytes between the start of rows in the buffer as allocated. If not given the value from `format_stride_for_width(format, width)` is used.

Returns a new *ImageSurface*

Raises *MemoryError* in case of no memory.

`cairo.Error` in case of invalid *stride* value.

Creates an *ImageSurface* for the provided pixel data. The initial contents of buffer will be used as the initial image contents; you must explicitly clear the buffer, using, for example, cairo_rectangle() and cairo_fill() if you want it cleared.

Note that the *stride* may be larger than width*bytes_per_pixel to provide proper alignment for each pixel and row. This alignment is required to allow high-performance rendering within cairo. The correct way to obtain a legal stride value is to call `format_stride_for_width()` with the desired format and maximum image width value, and use the resulting stride value to allocate the data and to create the *ImageSurface*. See `format_stride_for_width()` for example code.

classmethod **`create_from_png`** (*fobj*)

Parameters **fobj** – a filename, file, or file-like object of the PNG to load.

Returns a new *ImageSurface* initialized the contents to the given PNG file.

static **`format_stride_for_width`** (*format*, *width*)

Parameters

- **format** – a cairo *FORMAT* value
- **width** – the desired width of an *ImageSurface* to be created.

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.

Return type int

This method provides a stride value that will respect all alignment requirements of the accelerated image-rendering code within cairo. Typical usage will be of the form:

```
stride = cairo.ImageSurface.format_stride_for_width (format, width)
surface = cairo.ImageSurface.create_for_data (data, format, width, heigh
```

New in version 1.6.

get_data()

Returns a Python buffer object for the data of the *ImageSurface*, for direct inspection or modification.

New in version 1.2.

get_format()

Returns the *FORMAT* of the *ImageSurface*.

New in version 1.2.

get_height()

Returns the height of the *ImageSurface* in pixels.

get_stride()

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

get_width()

Returns the width of the *ImageSurface* in pixels.

2.7.3 class PDFSurface(Surface)

The PDFSurface is used to render cairo graphics to Adobe PDF files and is a multi-page vector surface backend.

class cairo.**PDFSurface**(*fobj*, *width_in_points*, *height_in_points*)

Parameters

- **fobj** (*None, str, unicode, file or file-like object*) – a filename or writable file object. None may be used to specify no output. This will generate a *PDFSurface* that may be queried and used as a source, without generating a temporary file.
- **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)

Returns a new *PDFSurface* of the specified size in points to be written to *fobj*.

Raises *MemoryError* in case of no memory

New in version 1.2.

set_size()

Parameters

- **width_in_points** (*float*) – new surface width, in points (1 point == 1/72.0 inch)
- **height_in_points** (*float*) – new surface height, in points (1 point == 1/72.0 inch)

Changes the size of a *PDFSurface* for the current (and subsequent) pages.

This function 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 function immediately after creating the surface or immediately after completing a page with either `Context.show_page()` or `Context.copy_page()`. New in version 1.2.

2.7.4 class PSSurface(Surface)

The *PSSurface* is used to render cairo graphics to Adobe PostScript files and is a multi-page vector surface backend.

class cairo.**PSSurface**(*fobj*, *width_in_points*, *height_in_points*)

Parameters

- **fobj** (*None, str, unicode, file or file-like object*) – a filename or writable file object. None may be used to specify no output. This will generate a *PSSurface* that may be queried and used as a source, without generating a temporary file.
- **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)

Returns a new *PDFSurface* of the specified size in points to be written to *fobj*.

Raises *MemoryError* in case of no memory

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

dsc_begin_page_setup()

This method indicates that subsequent calls to `dsc_comment()` should direct comments to the PageSetup section of the PostScript output.

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

See `dsc_comment()` for more details. New in version 1.2.

dsc_begin_setup()

This function indicates that subsequent calls to `dsc_comment()` should direct comments to the Setup section of the PostScript output.

This function 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. New in version 1.2.

dsc_comment(*comment*)

Parameters comment (*str*) – a comment string to be emitted into the PostScript output

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 characters. Violating either of these conditions will place *PSSurface* into an error state. But beyond these two conditions, this function 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 function 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 function 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 function 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 function 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 `Context.show_page()` or

`Context.copy_page()` comments are unambiguously directed to the PageSetup section of the current page. But it doesn't hurt to call this function 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

Here is an example sequence showing how this function might be used:

```
surface = PSSurface (filename, width, height)
...
surface.dsc_comment (surface, "%%Title: My excellent document")
surface.dsc_comment (surface, "%%Copyright: Copyright (C) 2006 Cairo Lov
...
surface.dsc_begin_setup (surface)
surface.dsc_comment (surface, "%%IncludeFeature: *MediaColor White")
...
surface.dsc_begin_page_setup (surface)
surface.dsc_comment (surface, "%%IncludeFeature: *PageSize A3")
surface.dsc_comment (surface, "%%IncludeFeature: *InputSlot LargeCapacit
surface.dsc_comment (surface, "%%IncludeFeature: *MediaType Glossy")
surface.dsc_comment (surface, "%%IncludeFeature: *MediaColor Blue")
... draw to first page here ..
ctx.show_page (cr)
...
surface.dsc_comment (surface, "%%IncludeFeature:  PageSize A5");
...
```

New in version 1.2.

get_eps()

Returns True iff the *PSSurface* will output Encapsulated PostScript.

New in version 1.6.

static **ps_level_to_string**(*level*)

Parameters **level** – a *PS_LEVEL*

Returns the string associated to given level.

Return type str

Raises `cairo.Error` if *level* isn't valid.

Get the string representation of the given *level*. See `ps_get_levels()` for a way to get the list of valid level ids. New in version 1.6.

restrict_to_level (*level*)

Parameters **level** – a *PS_LEVEL*

Restricts the generated PostSript file to *level*. See `ps_get_levels()` for a list of available level values that can be used here.

This function 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 function immediately after creating the surface. New in version 1.6.

set_eps (*eps*)

Parameters **eps** (*bool*) – True to output EPS format PostScript

If *eps* is True, the PostScript surface will output Encapsulated PostScript.

This function 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 function immediately after creating the surface. An Encapsulated PostScript file should never contain more than one page. New in version 1.6.

set_size (*width_in_points*, *height_in_points*)

Parameters

- **width_in_points** (*float*) – new surface width, in points (1 point == 1/72.0 inch)
- **height_in_points** (*float*) – new surface height, in points (1 point == 1/72.0 inch)

Changes the size of a PostScript surface for the current (and subsequent) pages.

This function 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 function immediately after creating the surface or immediately after completing a page with either `Context.show_page()` or `Context.copy_page()`. New in version 1.2.

2.7.5 class SVGSurface(`Surface`)

The *SVGSurface* is used to render cairo graphics to SVG files and is a multi-page vector surface backend

class `cairo.`**SVGSurface** (*fobj*, *width_in_points*, *height_in_points*)

Parameters

- **fobj** (*None, str, unicode, file or file-like object*) – a filename or writable file object. None may be used to specify no output. This will generate a *SVGSurface* that may be queried and used as a source, without generating a temporary file.
- **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)

Returns a new *SVGSurface* of the specified size in points to be written to *fobj*.

Raises *MemoryError* in case of no memory

get_versions()
Not implemented in pycairo (yet)

restrict_to_version()
Not implemented in pycairo (yet)

version_to_string()
Not implemented in pycairo (yet)

2.7.6 class Win32Surface(Surface)

The Microsoft Windows surface is used to render cairo graphics to Microsoft Windows windows, bitmaps, and printing device contexts.

class cairo.**Win32Surface**(*hdc*)

Parameters **hdc** (*int*) – the DC to create a surface for

Returns the newly created surface

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 resulting surface will always be of format cairo.FORMAT_RGB24, see *FORMAT attributes*.

2.7.7 class Win32PrintingSurface(Surface)

The Win32PrintingSurface is a multi-page vector surface type.

class cairo.**Win32PrintingSurface**(*hdc*)

Parameters **hdc** (*int*) – the DC to create a surface for

Returns the newly created surface

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; `show_page()` and associated methods must be used for correct output.

2.7.8 class XCBSurface(Surface)

The XCB surface is used to render cairo graphics to X Window System windows and pixmaps using the XCB library.

Note that the XCB surface automatically takes advantage of the X render extension if it is available.

class cairo.**XCBSurface**

Parameters

- **connection** – an XCB connection
- **drawable** – a X drawable
- **visualtype** – a X visualtype
- **width** – The surface width
- **height** – The surface height

Creates a cairo surface that targets the given drawable (pixmap or window).

Note: This methods works using xpyb.

---

set_size(*width*, *height*)

Parameters

- **width** – The width of the surface
- **height** – The height of the surface

Informs cairo of the new size of the X Drawable underlying the surface. For a surface created for a Window (rather than a Pixmap), this function must be called each time the size of the window changes. (For a subwindow, you are normally resizing the window yourself, but for a toplevel window, it is necessary to listen for ConfigureNotify events.)

A Pixmap can never change size, so it is never necessary to call this function on a surface created for a Pixmap.

2.7.9 class XlibSurface(Surface)

The XLib surface is used to render cairo graphics to X Window System windows and pixmaps using the XLib library.

Note that the XLib surface automatically takes advantage of X render extension if it is available.

class cairo.**XlibSurface**

Note: *XlibSurface* cannot be instantiated directly because Python interaction with Xlib would require open source Python bindings to Xlib which provided a C API. However, an *XlibSurface* instance can be returned from a function call when using pygtk http://www.pygtk.org/.

---

get_depth()

Returns the number of bits used to represent each pixel value.

New in version 1.2.

get_height()

Returns the height of the X Drawable underlying the surface in pixels.

New in version 1.2.

get_width()

Returns the width of the X Drawable underlying the surface in pixels.

New in version 1.2.

2.8 Text

Cairo has two sets of text rendering capabilities:

- The functions with text in their name form cairo's toy text API. The toy API takes UTF-8 encoded text and is limited in its functionality to rendering simple left-to-right text with no advanced features. That means for example that most complex scripts like Hebrew, Arabic, and Indic scripts are out of question. No kerning or correct positioning of diacritical marks either. The font selection is pretty limited too and doesn't handle the case that the selected font does not cover the characters in the text. This set of functions are really that, a toy text API, for testing and demonstration purposes. Any serious application should avoid them.
- The functions with glyphs in their name form cairo's low-level text API. The low-level API relies on the user to convert text to a set of glyph indexes and positions. This is a very hard problem and is best handled by external libraries, like the pangocairo that is part of the Pango text layout and rendering library. Pango is available from http://www.pango.org/.

2.8.1 class FontFace()

A *cairo.FontFace* specifies all aspects of a font other than the size or font matrix (a font matrix is used to distort a font by sheering it or scaling it unequally in the two directions). A *FontFace* can be set on a `Context` by using `Context.set_font_face()` the size and font matrix are set with `Context.set_font_size()` and `Context.set_font_matrix()`.

There are various types of *FontFace*, depending on the font backend they use.

class cairo.**FontFace**

Note: This class cannot be instantiated directly, it is returned by Context.get_font_face().

2.8.2 class FreeTypeFontFace(FontFace)

FreeType Fonts - Font support for FreeType.

The FreeType font backend is primarily used to render text on GNU/Linux systems, but can be used on other platforms too.

Note: FreeType Fonts are not implemented in pycairo because there is no open source Python bindings to FreeType (and fontconfig) that provides a C API. This a possible project idea for anyone interested in adding FreeType support to pycairo.

2.8.3 class ToyFontFace(FontFace)

The *cairo.ToyFontFace* class can be used instead of Context.select_font_face() to create a toy font independently of a context.

class cairo.**ToyFontFace** (*family*[, *slant*[, *weight*]])

Parameters

- **family** (*str or unicode*) – a font family name
- **slant** – the *FONT_SLANT* of the font, defaults to cairo.FONT_SLANT_NORMAL.
- **weight** – the *FONT_WEIGHT* of the font, defaults to cairo.FONT_WEIGHT_NORMAL.

Returns a new *ToyFontFace*

Creates a *ToyFontFace* from a triplet of family, slant, and weight. These font faces are used in implementation of the the "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 function for limitations of toy font faces. New in version 1.8.4.

get_family()

Returns the family name of a toy font

Return type str

New in version 1.8.4.

get_slant()

Returns the *FONT_SLANT* value

New in version 1.8.4.

get_weight()

Returns the *FONT_WEIGHT* value

New in version 1.8.4.

2.8.4 class UserFontFace(FontFace)

The user-font feature allows the cairo user to provide drawings for glyphs in a font. This is most useful in implementing fonts in non-standard formats, like SVG fonts and Flash fonts, but can also be used by games and other application to draw "funky" fonts.

Note: UserFontFace support has not (yet) been added to pycairo. If you need this feature in pycairo register your interest by sending a message to the cairo mailing list, or by opening a pycairo bug report.

2.8.5 class ScaledFont()

A *ScaledFont* is a font scaled to a particular size and device resolution. A *ScaledFont* is most useful for low-level font usage where a library or application wants to cache a reference to a scaled font to speed up the computation of metrics.

There are various types of scaled fonts, depending on the font backend they use.

class cairo.**ScaledFont**(*font_face*, *font_matrix*, *ctm*, *options*)

Parameters

- **font_face** – a FontFace instance
- **font_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. See Context.set_font_matrix().
- **ctm** – user to device transformation Matrix with which the font will be used.
- **options** – a FontOptions instance to use when getting metrics for the font and rendering with it.

Creates a *ScaledFont* object from a *FontFace* and matrices that describe the size of the font and the environment in which it will be used.

extents()

Returns (ascent, descent, height, max_x_advance, max_y_advance), a tuple of float values.

Gets the metrics for a *ScaledFont*.

get_ctm()
Not implemented in pycairo (yet)

get_font_face()

Returns the `FontFace` that this *ScaledFont* was created for.

New in version 1.2.

get_font_matrix()
Not implemented in pycairo (yet)

get_font_options()
Not implemented in pycairo (yet)

get_scale_matrix()

Returns the scale `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. New in version 1.8.

glyph_extents()
Not implemented in pycairo (yet)

text_extents(*text*)

Parameters **text** (*str or unicode*) – text

Returns (x_bearing, y_bearing, width, height, x_advance, y_advance)

Return type 6-tuple of float

Gets the extents for a string of text. The extents describe a user-space rectangle that encloses the "inked" portion of the text drawn at the origin (0,0) (as it would be drawn by `Context.show_text()` if the cairo graphics state were set to the same font_face, font_matrix, ctm, and font_options as *ScaledFont*). Additionally, the x_advance and y_advance values indicate the amount by which the current point would be advanced by `Context.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. New in version 1.2.

text_to_glyphs()
Not implemented in pycairo (yet)

2.8.6 class FontOptions()

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

Individual features of a *FontOptions* can be set or accessed using functions named *FontOptions.set_<feature_name>* and *FontOptions.get_<feature_name>*, like `FontOptions.set_antialias()` and `FontOptions.get_antialias()`.

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

class `cairo.`**`FontOptions`**

Returns a newly allocated *FontOptions*.

Allocates a new *FontOptions* object with all options initialized to default values.

`get_antialias``()`

Returns the *ANTIALIAS* mode for the *FontOptions* object

`get_hint_metrics``()`

Returns the *HINT METRICS* mode for the *FontOptions* object

`get_hint_style``()`

Returns the *HINT STYLE* for the *FontOptions* object

`get_subpixel_order``()`

Returns the *SUBPIXEL_ORDER* for the *FontOptions* object

`set_antialias``(`*antialias*`)`

Parameters **antialias** – the *ANTIALIAS* mode

This specifies the type of antialiasing to do when rendering text.

`set_hint_metrics``(`*hint_metrics*`)`

Parameters **hint_metrics** – the *HINT METRICS* mode

This controls whether metrics are quantized to integer values in device units.

`set_hint_style``(`*hint_style*`)`

Parameters **hint_style** – the *HINT STYLE*

This controls whether to fit font outlines to the pixel grid, and if so, whether to optimize for fidelity or contrast.

`set_subpixel_order``(`*subpixel_order*`)`

Parameters **subpixel_order** – the *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 `cairo.ANTIALIAS_SUBPIXEL`.

CHAPTER

THREE

FAQ

3.1 Pycairo FAQ - Frequently Asked Questions

Q: Can I subclass pycairo classes?

A: Cairo, the C library, is not an object oriented library, so a Python binding can never be a truly object oriented interface to cairo. One way to write the Python bindings for cairo would be as a single long list of module functions - this would be the most accurate representation of the underlying C library. Pycairo (and most other cairo language bindings?) instead chose to implement the bindings using Context, Surface, Pattern, etc classes. An advantage is that the classes organise cairo into groups of similar functions. A disadvantage is that creates an illusion that cairo is object oriented library, and people are then tempted to create subclasses to override cairo methods. When in fact there are no methods to override, just cairo functions which can't be overridden.

The cairo documentation Appendix A "Creating a language binding for cairo" section "Memory Management" describes why deriving from a Surface creates problems and is best avoided.

cairo.Context can be subclassed. All other pycairo subclasses cannot be subclassed.

Q: How do I use pycairo with numpy?

A: See test/isurface_create_for_data2.py

Q: How do I use pycairo with pygame?

A: See test/pygame-test1.py test/pygame-test2.py

CHAPTER

FOUR

PYCAIRO C API

This manual documents the API used by C and C++ programmers who want to write extension modules that use pycairo.

4.1 To access the Pycairo C API

Edit the client module file to add the following lines:

```
/* All function, type and macro definitions needed to use the Pycairo/C API
 * are included in your code by the following line
 */
#include "Pycairo.h"

/* define a variable for the C API */
static Pycairo_CAPI_t *Pycairo_CAPI;

/* import pycairo - add to the init<module> function */
Pycairo_IMPORT;
```

4.2 Pycairo Objects

Objects:

```
PycairoContext
PycairoFontFace
PycairoToyFontFace
PycairoFontOptions
PycairoMatrix
PycairoPath
PycairoPattern
PycairoSolidPattern
PycairoSurfacePattern
PycairoGradient
PycairoLinearGradient
PycairoRadialGradient
```

```
PycairoScaledFont
PycairoSurface
PycairoImageSurface
PycairoPDFSurface
PycairoPSSurface
PycairoSVGSurface
PycairoWin32Surface
PycairoXCBSurface
PycairoXlibSurface
```

4.3 Pycairo Types

Types:

```
PyTypeObject *Context_Type;
PyTypeObject *FontFace_Type;
PyTypeObject *ToyFontFace_Type;
PyTypeObject *FontOptions_Type;
PyTypeObject *Matrix_Type;
PyTypeObject *Path_Type;
PyTypeObject *Pattern_Type;
PyTypeObject *SolidPattern_Type;
PyTypeObject *SurfacePattern_Type;
PyTypeObject *Gradient_Type;
PyTypeObject *LinearGradient_Type;
PyTypeObject *RadialGradient_Type;
PyTypeObject *ScaledFont_Type;
PyTypeObject *Surface_Type;
PyTypeObject *ImageSurface_Type;
PyTypeObject *PDFSurface_Type;
PyTypeObject *PSSurface_Type;
PyTypeObject *SVGSurface_Type;
PyTypeObject *Win32Surface_Type;
PyTypeObject *XCBSurface_Type;
PyTypeObject *XlibSurface_Type;
```

4.4 Functions

cairo_t * **PycairoContext_GET**(obj)
get the C cairo_t * object out of the PycairoContext *obj

PyObject * **PycairoContext_FromContext**(cairo_t **ctx*, PyTypeObject **type*, PyObject **base*)

PyObject * **PycairoFontFace_FromFontFace**(cairo_font_face_t **font_face*)

PyObject * **PycairoFontOptions_FromFontOptions**(cairo_font_options_t **font_options*)

PyObject * **PycairoMatrix_FromMatrix**(const cairo_matrix_t **matrix*)

PyObject * **PycairoPath_FromPath**(cairo_path_t **path*)

PyObject * **PycairoPattern_FromPattern**(cairo_pattern_t **pattern*, PyObject **base*)

PyObject * **PycairoScaledFont_FromScaledFont**(cairo_scaled_font_t **scaled_font*)

PyObject * **PycairoSurface_FromSurface**(cairo_surface_t **surface*, PyObject **base*)

int **PycairoCheck_Status**(cairo_status_t *status*)

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *search*

PYTHON MODULE INDEX

C

INDEX

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X