

# Computer Graphics in Python



Advanced vector graphics  
using Pycairo and Python

---

Martin McBride

# Computer graphics in Python

Advanced vector graphics using Pycairo and Python

Martin McBride

This book is for sale at <http://leanpub.com/computergraphicsinpython>

This version was published on 2020-05-01



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2019 - 2020 Martin McBride

# Contents

<b>Foreword</b>	<b>1</b>
Who is this book for?	1
About the author	1
Keep in touch	1
<b>1 Introduction to vector graphics</b>	<b>3</b>
1.1 Pixel images	3
1.2 Size and resolution	4
1.3 Resizing pixel images	4
1.4 Drawing on pixel images	6
1.5 Vector graphics	7
1.6 Rendering	9
1.7 Typical uses	10
1.8 Benefits	10
1.9 Disadvantages	11
1.10 Common vector formats	12
<b>2 About Pycairo</b>	<b>14</b>
2.1 Cairo	14
2.2 Capabilities	14
2.3 Version	14
2.4 Installing Pycairo	14
2.5 Checking Pycairo version	15
<b>3 Basic drawing operations</b>	<b>16</b>
3.1 Creating an image with Pycairo	16
3.2 Coordinate system	19
3.3 Rectangles	20
3.4 Fill and stroke	21
3.5 Lines	23
3.6 Polygons	24
3.7 Open and closed shapes	25
3.8 Arcs	26
3.9 Circles	27

## CONTENTS

3.10 Bezier curves . . . . .	27
3.11 Line styles . . . . .	29
<b>4 Paths and complex shapes . . . . .</b>	<b>36</b>
4.1 Paths . . . . .	36
4.2 Sub-paths . . . . .	37
4.3 Lines . . . . .	39
4.4 Polygons . . . . .	41
4.5 Arcs . . . . .	43
4.6 Bezier curves . . . . .	50
4.7 Function curves . . . . .	54
4.8 Rectangle . . . . .	56
<b>5 Computer colour . . . . .</b>	<b>58</b>
5.1 RGB colour . . . . .	58
5.2 Pycairo RGB colours . . . . .	58
5.3 CSS named colours . . . . .	59
5.4 Transparency . . . . .	59
5.5 Transparency colour calculation . . . . .	59
5.6 Transparent images . . . . .	59
5.7 Greyscale images . . . . .	59
5.8 Pixel colours . . . . .	59
<b>6 Transforms and state . . . . .</b>	<b>60</b>
6.1 User space and device space . . . . .	60
6.2 Translation . . . . .	60
6.3 Scaling . . . . .	60
6.4 Rotation . . . . .	60
6.5 Save and restore . . . . .	61
6.6 Rotating about a point . . . . .	61
6.7 Placing an ellipse . . . . .	61
6.8 Correcting the effects of unequal scaling . . . . .	61
6.9 Flipping . . . . .	61
6.10 Current transform matrix . . . . .	61
<b>7 Working with text . . . . .</b>	<b>62</b>
7.1 Text is just shapes . . . . .	62
7.2 How Pycairo handles text . . . . .	62
7.3 Fonts . . . . .	64
7.4 Font size . . . . .	67
7.5 Font style . . . . .	68
7.6 Text extents . . . . .	70
7.7 Text extent examples . . . . .	71
7.8 Text alignment . . . . .	72

<b>8 Gradients and image fills</b>	<b>74</b>
8.1 Patterns	74
8.2 SolidPattern	74
8.3 Linear gradient	74
8.4 Linear gradients at different angles	75
8.5 Adding more stops	75
8.6 Extend options	75
8.7 Filling a stroke with a gradient	75
8.8 Filling text with a gradient	75
8.9 Radial gradients	76
8.10 Radial gradient with inner circle	76
8.11 Radial extend options	76
8.12 Loading an image into Pycairo	76
8.13 Using SurfacePattern with an image	76
8.14 SurfacePattern extend options	77
8.15 Using SurfacePattern with vectors	77
<b>9 Clipping, masking and compositing</b>	<b>78</b>
9.1 Clipping	78
9.2 Calling clip multiple times	78
9.3 Resetting the clip region	78
9.4 Clipping functions	78
9.5 Masking	78
9.6 Using an image as a mask	79
9.7 Compositing	79
9.8 OVER operator	79
9.9 Changing the drawing order	79
9.10 Masking operations	79
9.11 Artistic colour adjustments	80
9.12 Specific colour changes	80
<b>10 Surfaces and output formats</b>	<b>81</b>
10.1 Output formats	81
10.2 ImageSurface	81
10.3 SVGSurface	81
10.4 PDFSurface	82
10.5 PSSurface	82
10.6 RecordingSurface	82
10.7 ScriptSurface	82
10.8 TeeSurface	83
10.9 GUI surfaces	83
<b>11 Integration with other libraries</b>	<b>84</b>
11.1 How Pycairo stores image data	84

## CONTENTS

11.2 PIL (Pillow) integration . . . . .	85
11.3 Numpy integration . . . . .	85
<b>12 Reference . . . . .</b>	<b>86</b>
12.1 Radians . . . . .	86

# Foreword

The Pycairo library is a Python graphics library based on the Cairo library. Cairo is written in C and has been ported to many other languages.

PyCairo is an efficient, fully featured, high quality graphics library, with similar drawing capabilities to other vector libraries and languages such as SVG, PDF, HTML canvas and Java graphics.

Typical use cases include: standalone Python scripts to create an image, chart, or diagram; server side image creation for the web (for example a graph of share prices that updates hourly); desktop applications, particularly those that involve interactive images or diagrams.

The power of Pycairo, with the expressiveness of Python, is also a great combination for making *procedural images* such as mathematical illustrations and generative art. It is also quite simple to generate image sequences that can be converted to video or animated gifs.

## Who is this book for?

This book is primarily aimed at developers who have at least a small amount of Python experience, who are looking to use Python to create computer graphics for any application. The examples in the book use basic Python constructs, you don't need to be a Python guru to learn Pycairo. No prior knowledge of computer graphics is assumed, although if you have used other graphics systems you should still find this book useful as Pycairo has its own unique quirks and features.

## About the author

Martin McBride is a software developer, specialising in computer graphics, sound, and mathematical programming. He has been writing code since the 1980s in a wide variety of languages from assembler through to C++, Java and Python. He writes for PythonInformer.com and is the author of *Functional Programming in Python* available on leanpub.com. He is interested in generative art and works on the generativepy open source project.

## Keep in touch

If you have any comments or questions you can get in touch by any of the following methods:

- Joining the Python Informer forum at <http://pythoninformer.boards.net/><sup>1</sup>. Follow the [Computer Graphics in Python](#)<sup>2</sup> board in the Books section.

---

<sup>1</sup><http://pythoninformer.boards.net/>

<sup>2</sup><http://pythoninformer.boards.net/board/7/computer-graphics-python>

- The [book page](#)<sup>3</sup> on the Leanpub website.
- Signing up for the Python Informer newsletter at [pythoninformer.com](http://pythoninformer.com)
- Following [@pythoninformer](#) on twitter.
- Contacting me directly by email ([info@axlesoft.com](mailto:info@axlesoft.com)).

---

<sup>3</sup><https://leanpub.com/computergraphicsinpython>



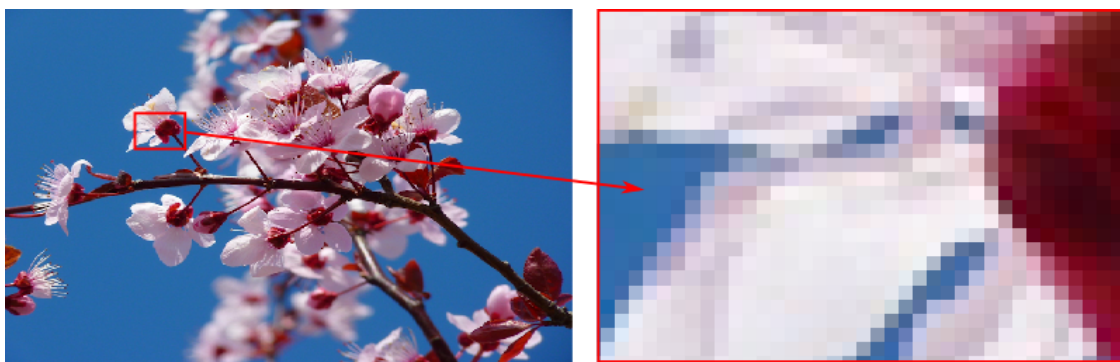
# 1 Introduction to vector graphics

There are two main ways to represent digital images - they can be stored as pixel images or vector images.

This book is about vector images, and more specifically how to use the Pycairo library to generate vector images. But we will start by looking at pixel images, because you can't fully understand one without the other.

## 1.1 Pixel images

A pixel image is an image that is made up of *pixels*. You can think of a pixel as being a tiny square area of the image. Each pixel is too small to see individually, but if you zoom right in you can see them, as this illustration shows:



The important thing about pixels is that each one represents a square of the image that is filled with a single colour. You can't have a pixel that is partly blue, partly red. Each of the squares in the zoomed image is completely filled with one colour. It is the combined effect of hundreds of tiny pixels that gives the illusion of a continuous image.

To store a pixel image, we simply need to store the colour value of each pixel, usually as an RGB (red, green, blue) value which typically occupies 3 bytes per pixel (1 byte per colour). Many modern cameras create images with millions, or tens of millions, of pixels, so pixel images can take a very large amount of storage space. Most common image formats, such as JPEG, PNG and GIF, employ data compression to reduce the storage requirement.

There are several alternative names for pixel images, which are more or less interchangeable:

- **Pixel image** - the word *pixel* is a shortened form of *picture element*, each pixel being an element of the overall image.

- **Raster image** - the word *raster* derived from the Latin for rake (*rastrum*). It refers to the way an image is drawn line by line, a bit like a rake being dragged over the ground.
- **Bitmap image** - in general a *bitmap* is a mapping between some kind of data and the bits in computer memory. This term was used to describe the mapping of computer memory onto pixels on the screen, which led to pixel images sometimes being called bitmap images.

## 1.2 Size and resolution

A key feature of a pixel image is that it has a fixed size in pixels. For example the image in the example above is 640 pixels wide and 400 pixels high.

The pixel size has an effect on the physical size of the image. The relationship between the pixel size and the physical size of the image is determined by the resolution it is printed or displayed at.

```
1 physical_size = pixel_size/resolution
```

Resolution is usually expressed as the number of pixels per inch (ppi). This is sometimes called dots per inch (dpi) or even lines per inch (lpi), they all mean the same thing. Resolution can also be expressed in centimetres or millimetres rather than inches (e.g. lpmm or lpm). We will use ppi.

For example, a typical computer monitor has a screen resolution of 90 ppi. If we display our 640 by 400 pixel image at 90 dpi, the physical size of the image on the screen will be about 7.1 by 4.4 inches (640 divided by 90, and 400 divided by 90).

Alternatively, if we printed the same image on a printer with a resolution of 300 ppi, our image would appear at a minuscule 2.1 by 1.3 inches.

The pixel size, physical size and resolution are linked by the formula above. If you choose any two of the values, the third value is fixed.

## 1.3 Resizing pixel images

It is *possible* to change the pixel size of an image, but there are costs and limitations.

You can make a pixel image smaller - for example you could take a 3000 by 2000 pixel image from a camera, and shrink it down to 600 by 400 pixels to fit on a website, or even to a 150 by 100 thumbnail image. Any image editing program such as Gimp or Photoshop will let you do that, quite successfully. The smaller image will look perfectly good. Some detail will be lost, simply because it has less pixels, but it won't be any worse than an image that had been created at that size in the first place.



It is also possible to keep the image at its original size, and scale it down when it is displayed. On a website, for example, you can use CSS to set the size display size of an image, and if it is too large the browser will scale it automatically. The downside is that you are storing an image that is bigger (in terms of the number of bytes) than you need to. If the image is being stored on disk it will take more space than it needs to, if it is being downloaded from a website it will take longer than it needs to.

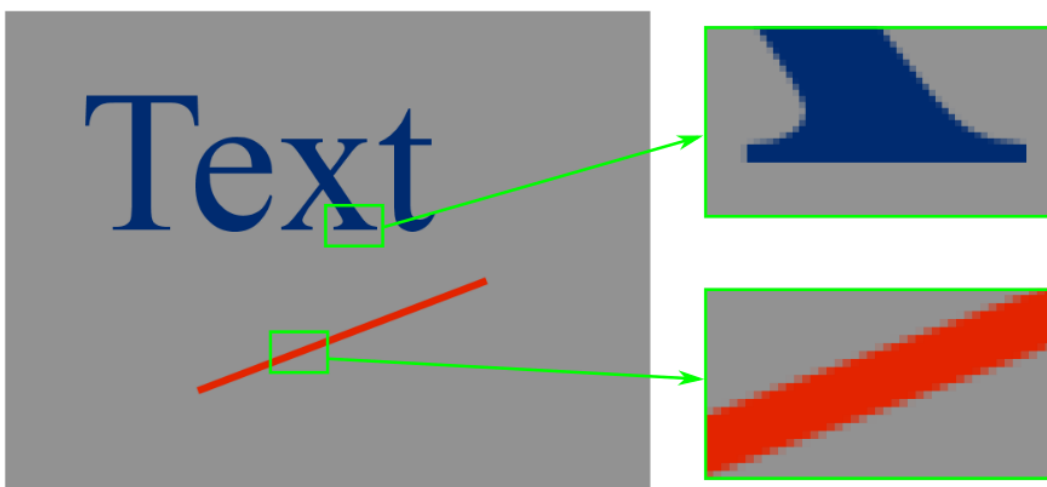
Trying to make a pixel image larger is usually less successful. A larger image contains more information than a small image. When you increase the pixel size of an image, you have no way of knowing what colour to make the extra pixels. If you wish to make the image slightly bigger, by 10% or maybe even 50%, the effect will be slight blurring. But if you try to increase the size more significantly, for example by a factor of 4, the result is a very poor quality image.



This illustrates a common problem with raster images - you need to decide the final pixel size of your image very early in the process of creating it.

## 1.4 Drawing on pixel images

You can draw shapes and text on pixel images. Most image editing software allows you to do that, and there are Python raster imaging libraries such as Pillow that allow you to do this in code. Painting shapes and text is simply a matter of changing the colour of the appropriate pixels:



There is nothing wrong with the text or the line as they are drawn, they are perfectly good quality. As you can see from the enlarged sections, the pixels close to the edges of the text and line have pixels that are a mix of the foreground and background colours - this is called *anti-aliasing* and it makes the edges appear smoother.

This technique suffers from a similar problem to the one mentioned above. At the point when you draw the text or the line, you are deciding the pixel size of your image. This is then baked in to text. If you later decide you want to print your image as a poster, requiring a much larger pixel size, you would pretty much need to start again.

There is another, more subtle, problem. When you draw on an image, you are overwriting the pixels you draw on. The original pixel colour is lost, so if there was something else behind the text when you draw it, that thing is gone forever.

This means, essentially, once you have draw something, you can't change it. You cant change the colour, the size, the font, or the position of the item, you can't even erase it altogether, because you don't know what was behind it before it was drawn.

Of course, many image editors have an undo feature, that allows you to undo what you just did, but that just works by keeping a temporary copy of the previous image to revert back to. As soon as you store your work as a PNG or JPEG file, the information is gone. Similarly, some packages have the idea of "layers" so that you can draw shapes an text on different layers, which can then be moved around, resized or deleted. These layers are really just separate images stored temporarily within the program, and again as soon as you save your work as an image file, the layers are merged into one, and can no longer be edited separately.

## 1.5 Vector graphics

Vector images take a totally different approach to storing images. Instead of storing the pixels that make up the image, a vector graphics file stores a list of instructions for how to draw the image.

Here is an example of an *SVG* file. SVG (Scalable Vector Graphics) is a widely used type of vector format that is based on XML. It is supported by all modern web browsers, and is the main file format used by the popular open source vector image editor Inkscape.

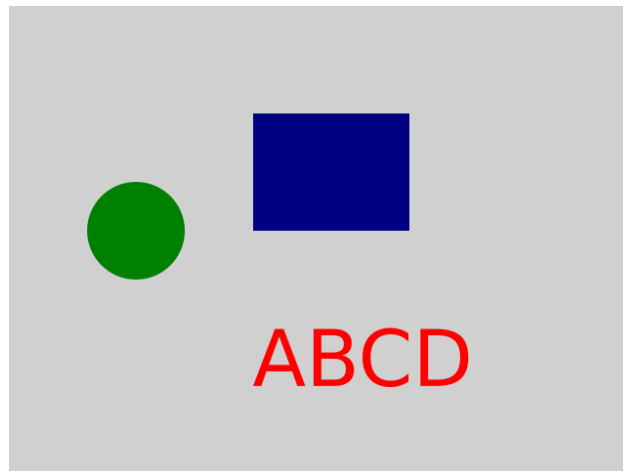
Don't worry about the details, you will probably never have to edit or even look at an SVG file in real life. This is just an illustration to give you an idea of what a vector graphics file might look like:

```

1  <svg
2    <g
3      id="layer1"
4      transform="translate(0,-572.36216)">
5    <circle
6      style="stroke:none;fill:#008000"
7      id="path5387"
8      cx="130"
9      cy="802.36218"
10     r="50" />
11    <rect
12      style="fill:#000080;stroke:none"
13      id="rect5389"
14      width="160"
15      height="120"
16      x="250"
17      y="682.36218" />
18    <text
19      xml:space="preserve"
20      style="font-style:normal;font-weight:normal;font-size:80px;line-height:125%;font-
21      ont-family:sans-serif;
22      letter-spacing:0px;word-spacing:0px;fill:#ff0000;fill-opacity:1;stroke:none"
23      x="249.375"
24      y="961.22937"
25      id="text5391"
26    ><tspan
27      id="tspan5393"
28      x="249.375"
29      y="961.22937">ABCD</tspan></text>
30  </g>
31 </svg>

```

Here is what the image looks like:



Even without understanding the format in any detail, it is fairly clear that it defines a circle, a rectangle and some text. For the circle it specifies a fill colour of #008000 (which is green in hex RGB), and gives the centre and radius values. For the text, it specifies the font, spacing and fill colour to use, as well as the actual text itself.

Some initial things to notice are:

- The file is entirely text based. There are no binary pixel values, or any other form of binary data.
- The data is resolution independent. For example the size and position of the circle (given by  $cx$ ,  $cy$  and  $r$ ) are simply floating point numbers. It specifies that the radius is 50 and the centre is (130, 802.36218), but depending on what units you choose this could be as big or as small as you like.
- This data is a complete specification of the vector image, given this file alone you can accurately and completely draw the image.

You could, if you wished, edit this image file with a normal text editor, to change the colours and sizes of the objects, or even change the text message itself. You can't do that easily with a pixel image of course.

## 1.6 Rendering

There is an aspect of vector graphics that is almost so obvious that you might miss it.

Most monitors and printers are pixel based. They cannot directly display a vector image. A vector image must be converted into a pixel image before it can be displayed or printed. This conversion is called *rendering* (or sometimes *rasterisation*).

This process is normally performed automatically by any software that supports vector formats. For example, if you open a PDF file using Adobe Acrobat or a similar product, it will automatically be

rendered for display on the monitor, so you can read or edit it. If you print the PDF file, it will be rendered by software in the printer (or possibly printer drivers installed on your computer) to be printed. If you open a web page that contains an SVG file, your browser will render the SVG to display it.

In each case the software will render the image at the correct resolution for the display or printer.

## 1.7 Typical uses

Typical uses of vector formats are for images that contains geometric shapes - diagrams, charts, graphs, logos and so on.

Vector formats are also great for storing text images. A text page is just a page full of letters, which if you think about it are just geometric shapes. Storing this as a vector image allows the text to be rendered properly on the screen or a printer.

A vector image can contain pixel images as elements of a page. In fact, it is quite common to have pixel images within a vector image. If you are reading this book as an ebook, it will be in a vector format such as PDF, ePub or similar. but it contains pixel images. However, some of the advantages of vector formats can be lost if they contain pixel images. When creating the document have to make a reasonable guess as to how many pixels the images should contains. The images in this book are sized to look reasonable on a screen or printed page, but if for any reason you wanted to print pages from this book at poster size, you would find the images somewhat pixelated.

Some vector formats can contain certain effects such as gradients, shadows, blurring. These effects are specified mathematically in the image, and are generated at the correct level of detail when the image is rendered.

## 1.8 Benefits

To summarise, here are the main benefits of of vector images:

- The image format is resolution independent. It can therefore be rendered at the most suitable resolution whenever it is printed or displayed.
- You can zoom into a vector image without losing detail. For example, if a vector image contained a page of small text, you could zoom in to make a single character fill the screen, and it would still be perfectly sharp.
- Since text is usually stored as text strings, you can search and extract text from an image.
- You can edit an image - adding and removing items, changing the size and colour of items, moving an item, etc.
- You can change the message in a text item.
- Vector images file sizes are often considerably smaller than the equivalent pixel image.



## 1.9 Disadvantages

Vector images have a few disadvantages compared to pixel images.

### 1.9.1 Complexity of conversion

A vector image requires quite complex processing to convert it to a pixel format for display or printing. When vector formats (mainly PostScript) first became popular in the 1980s, this presented a number of problems:

- It could take a long time to render a complex image from vector to pixel format.
- The conversion software was often proprietary, and sometimes expensive.
- Bugs were quite common both in the applications that created vector images, and the software that rendered them.

Such problems used to be quite serious. These days, thanks to modern computers that are orders of magnitude more powerful, and the existence of mature, open source libraries to deal with vector images (such as Pycairo), these problems are quite rare.

### 1.9.2 Pixel image elements

As mentioned earlier, vector formats are not ideal for any page elements that are based on pixel images. It isn't that they handle images particularly badly, but the presence of pixel images on a vector page means that you have to go back to worrying about the final intended output resolution for the image, which partly defeats the purpose of using vectors.

### 1.9.3 Fonts

A font is a file that represents a particular type face. The font file contains, in effect, a vector representation of every letter shape for that font.

Most vector formats define text features by supplying:

- The string representation of the text itself, such as "ABCD".
- The name of the font, such as Arial.

In order to render a vector image containing text, it needs access to the original font file - that is the only way it can find out what shape to make each letter.

This can create problems if you create a vector image and send it to someone who wants to view it on a different computer. If they do not have the correct fonts installed, a different one will be selected

from the fonts that are available on their system. It will hopefully have similar characteristics, but even so it will probably not look exactly the same.

It is often best to stick to fairly standard fonts that are likely to be present on most systems.

Some vector formats, such as PDF, allow you to include the font file as part of the vector image, so that any system will be able to render the file correctly. This can significantly increase the file size.

### 1.9.4 Editability

Vector files can be edited, including altering the wording of the text.

This is often very useful. In some cases, though, it is not desirable. For example, if you have a contract or agreement in PDF format, it certainly isn't a good thing that either party could change the words without anyone being able to prove which version is the correct one.

In these cases, one solution is to digitally sign the file.

## 1.10 Common vector formats

Here are some of the most commonly used or well known vector graphics formats.

**SVG** (Scalable Vector Graphics) is an open standard for vector graphics. It was developed by the World Wide Web Consortium (W3C) mainly for the use as a web format for vector graphics. It is an XML based format that can be directly embedded in HTML.

**PDF** (Portable Document Format) developed by Adobe to store documents in vector format. PDF also includes extra features such as interactive fields and annotations, attachments, encryption and digital signatures.

**PostScript** is an older standard developed by Adobe prior to PDF. It is unusual in that, rather than being a normal file format, PostScript is actually a programming language. A PostScript document is effectively a program that executes to draw the page. This idea often caused more problems than it solved - in particular the act of printing a page involved executing some code from a potentially unknown source, which is now considered to be a rather serious security risk.

**EPS** (Encapsulated PostScript) is a vector format using a subset of the PostScript language to define content that is designed to form a part of larger page. It is still used, but it shares some of the potential security vulnerabilities of PostScript.

**EMF** (Enhanced Metafile) is a Windows vector format for storing images that can be embedded in other documents. It can also be used as a graphics language for printers. It is quite Windows specific as its structures are closely related to Windows graphics library (GDI) calls. This again is a potential source of security vulnerabilities. EMF replaces the older **WMF** (Windows Metafile) format.

**XPS** (XML Paper Specification) is an open specification developed by Microsoft. It is similar to PDF in many ways, but information is stored in an XML based format rather than the PDF dictionary

structure. It provides static page descriptions, and by design it does not contain any of the PDF interactive features.

**MathML** (Mathematical Markup Language) is an XML based format for describing mathematical formulae. It is part of the HTML5 standard.

This is far from a comprehensive list, there are many other lesser known formats.

## 2 About Pycairo

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 2.1 Cairo

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 2.2 Capabilities

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 2.3 Version

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 2.4 Installing Pycairo

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 2.4.1 Windows

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 2.4.2 Linux

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 2.4.3 Mac

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 2.5 Checking Pycairo version

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 3 Basic drawing operations

In vector graphics, we don't directly paint individual pixels. Instead, we define shapes and tell Pycairo how we want them to be filled or outlined.

Our completed image can be stored as a vector image (such as SVG) or it can be converted into a raster format such as PNG. In most of the examples in this chapter we will create PNG files, but we will also see how to create other types of output.

Drawing shapes is a very important aspect of Pycairo programming, so we have devoted two chapters to the topic. In this chapter we will learn about the basic shape functions and how to use them to create simple images in Pycairo. In a later chapter we will revisit this and look at practical examples of creating more complex shapes.

### 3.1 Creating an image with Pycairo

We will start by creating a very simple PNG image, containing just a single rectangle. Here is the code:

```
1  import cairo
2
3  # Set up pycairo
4  surface = cairo.ImageSurface(cairo.FORMAT_RGB24, 600, 400)
5  ctx = cairo.Context(surface)
6  ctx.set_source_rgb(0.8, 0.8, 0.8)
7  ctx.paint()
8
9  # Draw the image
10 ctx.rectangle(150, 100, 100, 240)
11 ctx.set_source_rgb(1, 0, 0)
12 ctx.fill()
13
14 # Save the result
15 surface.write_to_png('rectangle.png')
```

Here is image this creates:



We will go through this code, step by step, to gain a better understanding of how Pycairo creates images.

### 3.1.1 Setting up Pycairo

You will need to have Python and Pycairo installed on your system (see the chapter *About Pycairo*). As with any Python module, you must import it before you can use it:

```
1 import cairo
```

Next, we must create a surface. A surface is the object where you create your image. You can think of it a bit like an artist's canvas:

```
1 surface = cairo.ImageSurface(cairo.FORMAT_RGB24, 600, 400)
```

When we create the surface, we are also deciding what sort of output we are going to produce:

- An ImageSurface will create a PNG image as output. There are other types of surface that create different output formats, for example SVG or PDF.
- We set the format to `FORMAT_RGB24`, which is a normal RGB image.
- We set the output image size to 600 by 400 pixels.

The next step is to create a *context*. A context is the thing we use to draw on the surface. You could think of it as being analogous to a pen that draws on the surface, but this is quite a loose analogy. Here is how we create a context:

```
1 ctx = cairo.Context(surface)
```

Finally, we will fill the entire surface with a light grey colour, using the `paint` function:

```
1 ctx.set_source_rgb(0.8, 0.8, 0.8)
2 ctx.paint()
```

The Context function `set_source_rgb` is used to set a colour. The colour is determined by 3 values that control the amount of red, green and blue in the colour. Values of 0 mean none of that colour is present, values of 1 means that the colour is as bright as possible. Setting the colour to (0.8, 0.8, 0.8) means that the red, green and blue values are each set to 80% of their maximum, which gives a light grey colour. See the chapter *Colour* for more details.

The `paint` function fills the entire surface with the previously selected colour. This is why the image (above) has a light grey background.

### 3.1.2 Drawing the image

This is the code that draws the red rectangle:

```
1 # Draw the image
2 ctx.rectangle(150, 100, 100, 240)
3 ctx.set_source_rgb(1, 0, 0)
4 ctx.fill()
```

Drawing in Pycairo is a two-step process. First, we define a *path*. A path is an abstract description of what we intend to draw. The second stage is to actually draw the shape described by the path.

The `rectangle` function adds a rectangle shape to the path. It doesn't draw a rectangle, it just adds it to the plan of what we are intending to draw. The function has the following definition:

```
1 rectangle(x, y, width, height)
```

It specifies a rectangle placed at position (x, y) with the given width and height. Our code will place a rectangle at position (150, 100), with a width of 100 and a height of 240. We will explain how coordinates work in a little while.

Next, we set our colour with `set_source_rgb`, this time to (1, 0, 0), which is 100% red with no green or blue – in other words, bright red.

Finally, we call `fill`. This function looks at the path we have defined and fills it with the colour we previously selected. The `fill` function is what actually draws the shape, in this case a rectangle.

### 3.1.3 Saving the PNG file

When we have finished drawing, we can save the image as a PNG file:



```
1 # Save the result
2 surface.write_to_png('rectangle.png')
```

This saves the image to the file *rectangle.png* in the current folder (usually wherever the Python source file is). Of course, you could add a full file path rather than just a filename if you wanted to write the file to a specific folder.

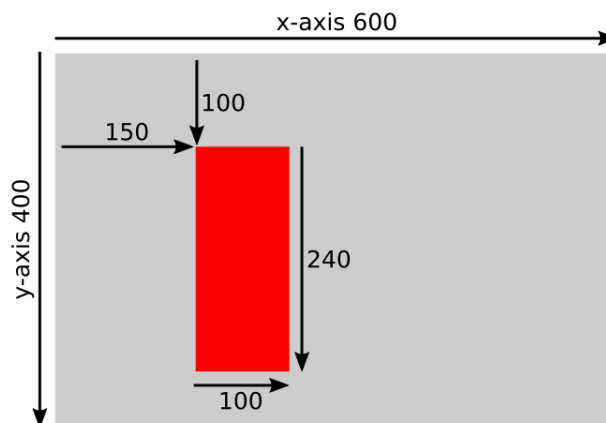
Notice that `write_to_png` is a function of the `Surface` object. This makes sense if you think about it, the surface is the object that owns the image.

Now that we know how to draw a simple image, we will learn a bit more about the way Pycairo defines the position and size of objects (ie the coordinate system), and how to draw a variety of other simple shapes.

For the rest of this chapter, all the examples will use the same basic code to draw a 600 by 400 pixel image. For brevity, we will only show the code that belongs in the *Draw the image* section of the code.

## 3.2 Coordinate system

By default, Pycairo uses a simple coordinate system based on the size of the output image you are creating (specified by the `cairo.ImageSurface` call). In the previous example, the image is 600 by 400 pixels in size, so the coordinate space is also 600 by 400 units:



Pycairo coordinates have the origin at the top left of the page, as do many other computer graphics libraries. This is different to the convention used in mathematics, where the origin is usually at the bottom left of the page.



A likely reason for this is that early computer monitors were based on Cathode Ray Tube (CRT) technology used by analogue TVs at the time. TVs scanned left to right, top to bottom – this might well have been an unconscious design decision at the time since most western languages are read in that order, so it was probably seen as the “natural” scanning order. Computer monitors adopted the same standard as TVs, computer video memory was organised to match the monitor scanning order, and graphics libraries were designed to match the arrangement of the data in video memory.

If we look again at the code that defines the red rectangle:

```
1 ctx.rectangle(150, 100, 100, 240)
```

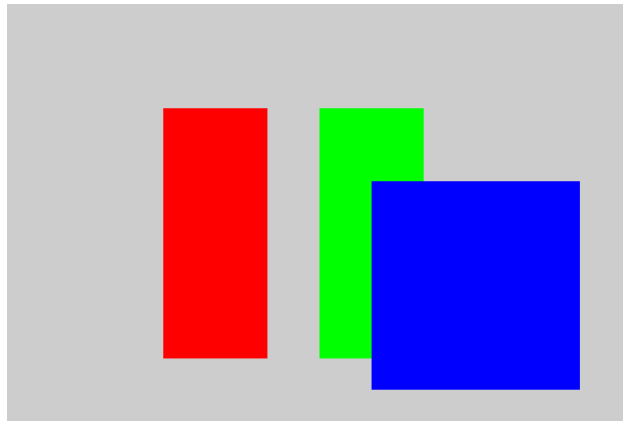
The rectangle is positioned at point (150, 100). This means that the top left corner of the rectangle is positioned 150 pixels to the right and 100 pixels down from the top left corner of the image. The rectangle itself is 100 pixels wide and 240 pixels high.

## 3.3 Rectangles

In this next example we will draw two rectangles and a square. Here is the code:

```
1 ctx.rectangle(150, 100, 100, 240)
2 ctx.set_source_rgb(1, 0, 0)
3 ctx.fill()
4
5 ctx.rectangle(300, 100, 100, 240)
6 ctx.set_source_rgb(0, 1, 0)
7 ctx.fill()
8
9 ctx.rectangle(350, 170, 200, 200)
10 ctx.set_source_rgb(0, 0, 1)
11 ctx.fill()
```

This code is similar to the previous example code, but we have extended it to draw 3 rectangles. In each case, we set the position and size (different for each rectangle), select the colour (a red, a green and a blue rectangle), and call fill to draw the rectangle.



One thing to notice is that the blue square partially overlaps the green rectangle. This is not uncommon in computer graphics. Pycairo handles this in a consistent way. Each new object is painted over the top of any existing objects. The green rectangle is painted first, then the blue square is painted on top of it.

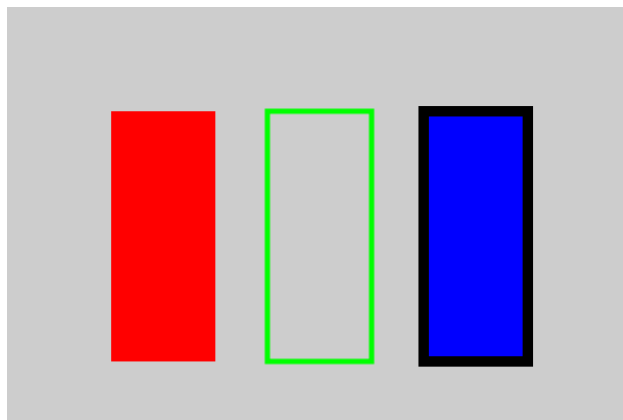


There are some advanced drawing techniques that can change the normal drawing order. These are called *compositing* operations, and are covered in a later chapter.

## 3.4 Fill and stroke

So far, we have drawn filled rectangles. Sometimes you might just want to draw the outline of a shape. This is called *stroking* (as in a stroke of the pen).

You can also fill and stroke a shape, usually with different colours. Here is an example of each case. The red shape on the left is filled, the green shape in the middle is stroked, the blue shape on the right is filled and stroked (in black).



Here is the code to draw the red filled shape. This is the same as the previous example:

```
1  # Fill
2  ctx.rectangle(100, 100, 100, 240)
3  ctx.set_source_rgb(1, 0, 0)
4  ctx.fill()
```

Here is the code to draw the green stroked shape:

```
1  # Stroke
2  ctx.rectangle(250, 100, 100, 240)
3  ctx.set_source_rgb(0, 1, 0)
4  ctx.set_line_width(5)
5  ctx.stroke()
```

We use the `rectangle` function, as normal, then use `set_source_rgb` to set the colour of the outline. There is then an extra step – we call `set_line_width` to choose how wide the outline should be. Line width is measured in the units of the coordinate system. The whole rectangle is 100 units wide, so a 5 unit outline gives a moderately thick outline. Calling the `stroke` function then draws the outline of the rectangle, using the required colour and thickness.

And here is how to draw the blue filled and stroked shape:

```
1  # Fill and stroke
2  ctx.rectangle(400, 100, 100, 240)
3  ctx.set_source_rgb(0, 0, 1)
4  ctx.fill_preserve()
5  ctx.set_source_rgb(0, 0, 0)
6  ctx.set_line_width(10)
7  ctx.stroke()
```

This is similar to the previous examples. We define the rectangle shape. Then we set the blue colour and fill the shape. Then we set the black colour, set the line width, and stroke the shape.

There is just one little wrinkle. We said earlier that the `rectangle` function adds a rectangle shape to the path, ready to be drawn. There is something else we need to know – the `fill` and `stroke` functions not only draw the path, they also clear the path afterwards.

This is usually what we want. Normally we define a path, then either fill it or stroke it, and then it the path has been automatically cleared, so we can carry on and draw the next shape.

However, if you want to fill and stroke a shape, you have an obvious problem. You create the path and fill it, but then the path is deleted so you can stroke it. The `fill_preserve` function does the same job as `fill`, but it *doesn't* clear the path. We can then use `stroke` to outline the same rectangle. We will look at this in a bit more detail in the chapter *Paths and complex shapes*.

## 3.5 Lines

So far, we have used rectangles to illustrate drawing in Pycairo. That is a good place to start because the `rectangle` function is a very convenient way to create a shape.

Generally, though, if you need to create any other type of shape, you have to construct it from separate lines and curves. In this section we will see how to draw a simple straight line. Here is the code we use to do this:

```
1 ctx.move_to(100, 100)
2 ctx.line_to(500, 300)
3 ctx.set_source_rgb(1, 0, 0)
4 ctx.set_line_width(10)
5 ctx.stroke()
```

And here is the image it creates:



Pycairo uses the idea of a *current point* when drawing shapes. This makes it easy to create common shapes, such as polygons, that are formed from lines or curves joined end-to-end.

Here is how it works when we draw a line (in the code above):

- Initially the path is empty and the current point is undefined.
- The `move_to` function sets the current point to (100, 100).
- The `line_to` function adds a line to the path from the current point to the specified point (500, 300). It also sets the current point to the new value (500, 300).

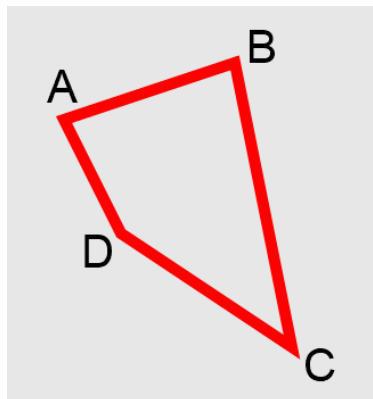
At this point, the path describes a line from (100, 100) to (500, 300). We then draw this line by setting the colour, setting the line width, and calling `stroke`. We can't fill a line, of course, because unlike a rectangle, a line doesn't enclose an area of the page.

## 3.6 Polygons

A polygon is closed shape composed of straight lines. Here is how we draw a simple polygon with 4 sides:

```
1 ctx.move_to(50, 100) # A
2 ctx.line_to(200, 50) # B
3 ctx.line_to(250, 300) # C
4 ctx.line_to(100, 200) # D
5 ctx.close_path()
6 ctx.set_source_rgb(1, 0, 0)
7 ctx.set_line_width(10)
8 ctx.stroke()
```

This draws the shape below:



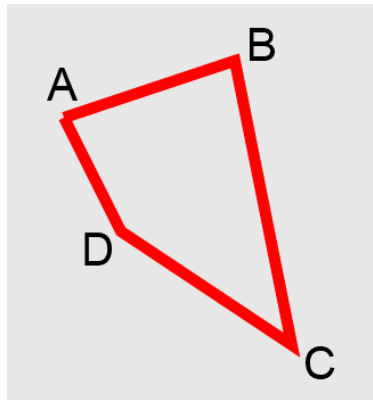
Here is the sequence of events:

- Initially, the path is empty and the current point is undefined.
- `move_to` sets the current point to A.
- `line_to` adds a line to the path, from the current point A to the point B. It also sets the current point to the new value B.
- `line_to` adds another line from B to C, and sets the current point C.
- `line_to` adds another line from C to D, and sets the current point D.
- `close_path` completes the closed shape. It adds a line from the current point D back to the first point in the path, which was A.

As before, we set the colour and line width, then stroke the path to draw the polygon.

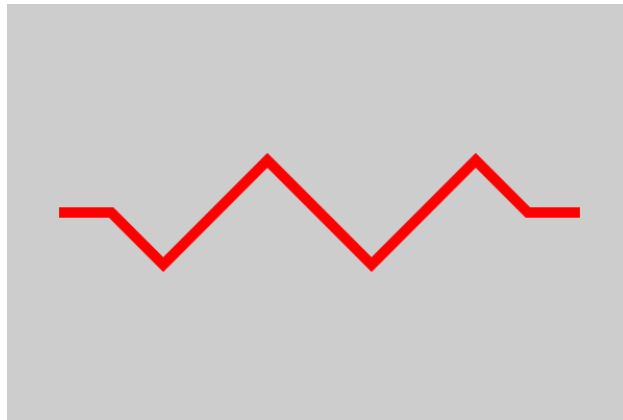
An important point here is that `close_path` actually *joins* point D to point A. It makes the corner look like a proper joined corner. Don't be tempted to use another `line_to` to join point D to point

A. It will draw a line, but it won't create a proper corner at A. This is the effect – the lines at corner A don't join correctly, making it different to the other corners:



## 3.7 Open and closed shapes

We can also create open shapes – a set of lines that join end-to-end but don't join start to end. This is called an open-polygon, or sometimes a polyline. Here is an example:



```
1 ctx.move_to(50, 200)
2 ctx.line_to(100, 200)
3 ctx.line_to(150, 250)
4 ctx.line_to(250, 150)
5 ctx.line_to(350, 250)
6 ctx.line_to(450, 150)
7 ctx.line_to(500, 200)
8 ctx.line_to(550, 200)
9 ctx.set_source_rgb(1, 0, 0)
10 ctx.set_line_width(10)
11 ctx.stroke()
```

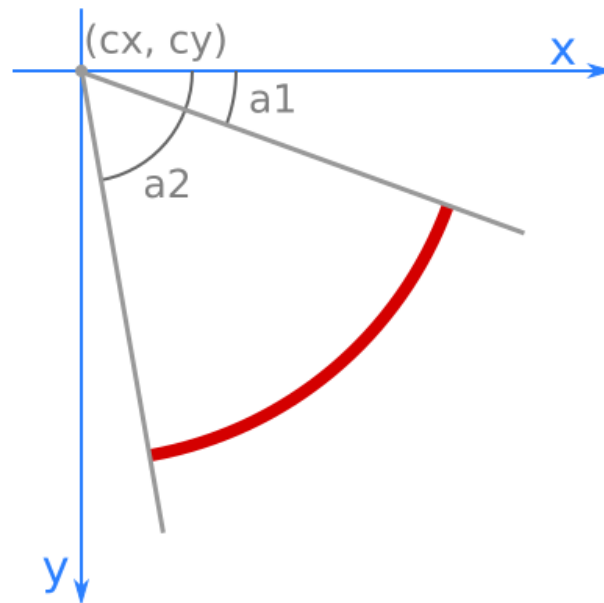
We create an open shape in the same way we create a closed shape, except we don't call `close_path` at the end. Open shapes can be used for decorative dividers, symbols, arrows and many other applications.

## 3.8 Arcs

An *arc* is a part the circumference of a circle. The `arc` function is called like this:

```
1 ctx.arc(cx, cy, r, a1, a2)
```

This code will create an arc like this:



The arc is part of a circle, radius  $r$ , centred at the point  $(cx, cy)$ . The arc starts at angle  $a1$  and ends at angle  $a2$ . Angles are measured from the x-axis, and by default increase as you move clockwise.

Here is an example of the code to draw an arc:

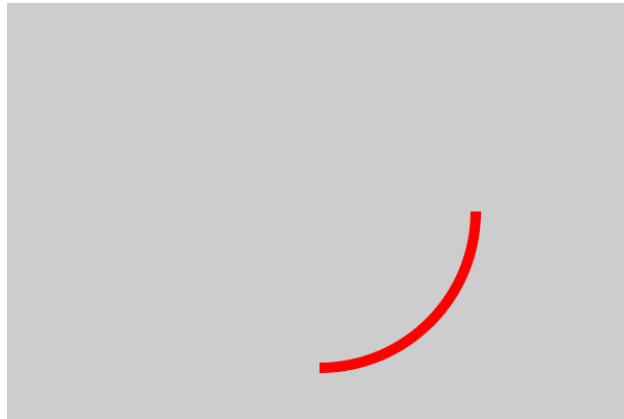
```
1 ctx.arc(300, 200, 150, 0, math.pi/2)
2 ctx.set_source_rgb(1, 0, 0)
3 ctx.set_line_width(10)
4 ctx.stroke()
```

Angles in Python, as in most computer languages, are measured in radians rather than degrees.  $\pi$  radians is exactly 180 degrees. That makes 1 radian equal to approximately 57.3 degrees.

For more information, see the *Reference* chapter at the end of the book.



The code above draws an arc of a circle, centre (300, 200) – the middle of the image. The circle radius is 150, and the arc goes from angle 0 to angle  $\pi/2$  radians, which is a quarter of a turn. The image looks like this:



## 3.9 Circles

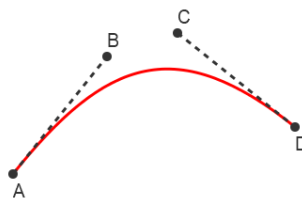
There is no function for drawing a circle. Instead, you can just use the `arc` function again. An arc from 0 to  $2\pi$  radians (which is equivalent to a full circle, 360 degrees) draws a complete circle.

Here is how to draw a circle, centre (cx, cy) and radius r:

```
1 ctx.arc(cx, cy, r, 0, 2*math.pi)
```

## 3.10 Bezier curves

A *Bezier curve* is a versatile curve that can be used to create many different shapes. It is based on four points. The anchors, A and D are the end points of the curve. The control points B and C control the route the curve takes between the anchors.

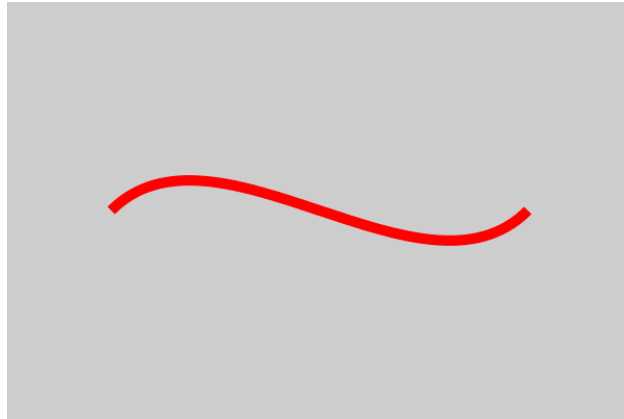


We will cover the properties of this curve in a later chapter, but for now we will just take a quick look at how it works. You might also try experimenting with a drawing package such as Inkscape, which allows you to create Bezier curves and move the 4 points around to try different curves.

Here is the code to draw a single Bezier curve:

```
1 ctx.move_to(100, 200)
2 ctx.curve_to(200, 100, 400, 300, 500, 200)
3 ctx.set_source_rgb(1, 0, 0)
4 ctx.set_line_width(10)
5 ctx.stroke()
```

Here is the curve it produces:



In the code, `move_to` sets the current point, in the usual way.

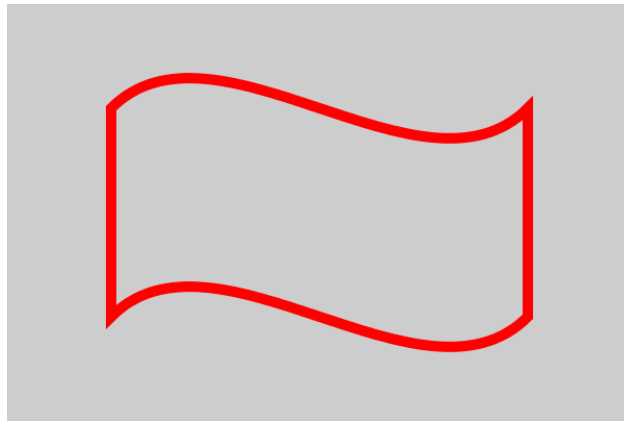
The `curve_to` function accepts six parameters. It treats the current point as the first anchor A. The six parameters represent the x, y values of the points B, C and D.

After calling `curve_to`, the current point is set to the location of D, which is the end of the curve.

You can also join curves and lines together, to form polycurves. Here we join a curve, a line, and another curve, then close the path (creating another line) to form a shape:

```
1 ctx.move_to(100, 100)
2 ctx.curve_to(200, 0, 400, 200, 500, 100)
3 ctx.line_to(500, 300)
4 ctx.curve_to(400, 400, 200, 200, 100, 300)
5 ctx.close_path()
6 ctx.set_source_rgb(1, 0, 0)
7 ctx.set_line_width(10)
8 ctx.stroke()
```

Here is the result:



## 3.11 Line styles

We will finish this chapter with a look at various ways you can change the appearance of stroked lines. These techniques apply to all stroked lines – single lines, polygons, arcs and Bezier curves.

You might find that you usually just use the defaults, but it is useful to know that other options exist.

### 3.11.1 Line caps

The line cap value affects how the ends of the lines are drawn. There are three options:



Here is the code to draw these lines:

```
1 ctx.set_source_rgb(0, 0, 0)
2 ctx.set_line_width(20)
3
4 ctx.move_to(100, 80)
5 ctx.line_to(500, 80)
6 ctx.set_line_cap(cairo.LINE_CAP_BUTT)
7 ctx.stroke()
8
9 ctx.move_to(100, 200)
10 ctx.line_to(500, 200)
11 ctx.set_line_cap(cairo.LINE_CAP_SQUARE)
12 ctx.stroke()
13
14 ctx.move_to(100, 320)
15 ctx.line_to(500, 320)
16 ctx.set_line_cap(cairo.LINE_CAP_ROUND)
17 ctx.stroke()
```

The lines are drawn in the normal way, except that we make an extra call to `set_line_cap` to set the line cap style for each line.

For illustration, red dots have been added to mark the exact positions of the points passed into the `line_to` and `move_to` functions.

The top option is `LINE_CAP_BUTT`. This is the default if you don't call `set_line_cap` at all. The line starts and ends exactly on the points specified, and the ends are squared off. This is useful if you want to ensure that your lines are exactly the right length.

The middle option is `LINE_CAP_SQUARE`. In this case, the line ends are squared off, but they extend slightly beyond the exact points. In fact, they extend by exactly half of the line width.

The bottom option is `LINE_CAP_ROUND`. In this case, the line ends are semi-circles, that extend beyond the exact points. The radius of the semi-circle is exactly half the line width.

The choice is largely dependent the visual effect you are wishing to achieve. The line cap style also affects line dashes. You might also prefer to match it with the line join style.

### 3.11.2 Line joins

You can also control how lines join. Here are the three possible styles:



Here is the code to draw these examples:

```
1  ctx.set_source_rgb(0, 0, 0)
2  ctx.set_line_width(20)
3
4  ctx.move_to(50, 100)
5  ctx.line_to(180, 300)
6  ctx.line_to(50, 300)
7  ctx.set_line_join(cairo.LINE_JOIN_MITER)
8  ctx.stroke()
9
10 ctx.move_to(240, 100)
11 ctx.line_to(370, 300)
12 ctx.line_to(240, 300)
13 ctx.set_line_join(cairo.LINE_JOIN_ROUND)
14 ctx.stroke()
15
16 ctx.move_to(430, 100)
17 ctx.line_to(560, 300)
18 ctx.line_to(430, 300)
19 ctx.set_line_join(cairo.LINE_JOIN_BEVEL)
20 ctx.stroke()
```

We use the `set_line_join` function to set the style.

The example on the left is `LINE_JOIN_MITER`. This is the default if you don't call `set_line_join` at all. This style creates a pointed corner, similar to a mitre joint used in carpentry.

The middle example is `LINE_JOIN_ROUND`. This rounds off the point at the corner.

Finally, on the right is `LINE_JOIN_BEVEL`. This removes the point at the corner with a straight cut.

Notice that these effects only occur where lines actually join, that is:

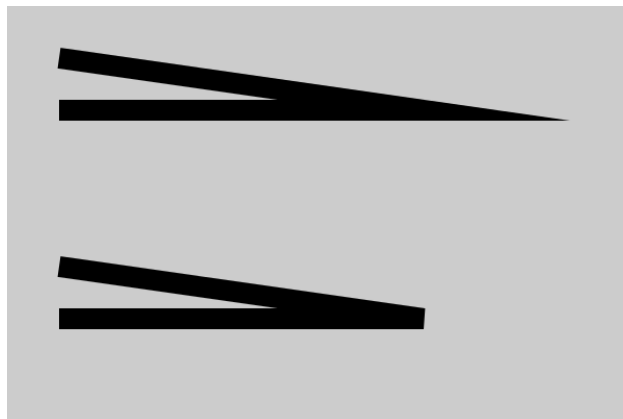
- The lines are linked by chained calls to `line_to`, `curve_to` or `arc` (similar to the polygons and polycurves we drew earlier).
- The lines is created by `close_path`, which joins the end point back to the start point in a closed shape.

### 3.11.3 Mitre limit

At this point we will briefly mention the mitre limit. One unfortunate side effect of the mitre join is that at very narrow angles, the length of the mitre point can get stupidly long. In fact, as the angle gets closer to zero, the spike gets longer and longer, without limit!

This isn't a bug, it is just how mitres work. But it can be quite undesirable, because the spike can start encroaching into parts of the page where it isn't really wanted.

To avoid this, Pycairo has a mitre limit. If the angle is less than 11 degrees, it automatically uses bevel mode instead of mitre mode. Here is an example, the top join has mitre limit disabled, the bottom one has mitre limit enabled:



You don't usually need to worry about the mitre limit, it is enabled automatically and is generally a good thing to have. If you really need to change it, you can use the function:

```
1 ctx.set_miter_limit(limit)
```

The limit value determines when the mitre limit applies. In general, if you want the mitre limit to apply at angle  $a$  or below, use the following limit value:

```
1 limit = 1/math.sin(a/2)
```

Here are some examples:

- Limit value 1.414 (the square root of 2) applies a mitre limit at 90 degrees or less
- Limit value 2 applies a mitre limit at 60 degrees or less
- Limit value 10 (the default) applies a mitre limit at 11 degrees or less

### 3.11.4 Dashed lines

Dashed lines are very useful in diagrams and charts. They provide a familiar and intuitive way to distinguish different types of information. For example, if you wanted to add an annotation to a graph, you might enclose it in a box with a dashed outline to indicate that it is additional information, not part of the graph itself.

Dashed lines can also be used purely for decoration.

Of course, drawing a dashed line could potentially be quite complicated, if you needed to draw each dash separately. Fortunately, Pycairo can create a variety of dashed automatically. All you need to do is define the dash pattern, using the `set_dash` function, like this:

```
1  ctx.set_source_rgb(0, 0, 0)
2  ctx.set_line_width(5)
3
4  ctx.move_to(100, 50) #1
5  ctx.line_to(500, 50)
6  ctx.set_dash([20])
7  ctx.stroke()
8
9  ctx.move_to(100, 100) #2
10 ctx.line_to(500, 100)
11 ctx.set_dash([20, 10])
12 ctx.stroke()
13
14 ctx.move_to(100, 150) #3
15 ctx.line_to(500, 150)
16 ctx.set_dash([20, 5, 5, 5])
17 ctx.stroke()
18
19 ctx.move_to(100, 200) #4
20 ctx.line_to(500, 200)
21 ctx.set_dash([5, 5, 10])
22 ctx.stroke()
23
24 ctx.set_line_width(10)
25 ctx.set_line_cap(cairo.LINE_CAP_ROUND)
26
27 ctx.move_to(100, 250) #5
28 ctx.line_to(500, 250)
29 ctx.set_dash([10, 20])
30 ctx.stroke()
31
```

```

32 ctx.move_to(100, 300) #6
33 ctx.line_to(500, 300)
34 ctx.set_dash([0, 20])
35 ctx.stroke()

```

This code draws six lines, with different dash patterns. The first four are 5 units wide and use the default butt line cap. The last two are 10 units wide and use the round line cap. They all illustrate different dash patterns:



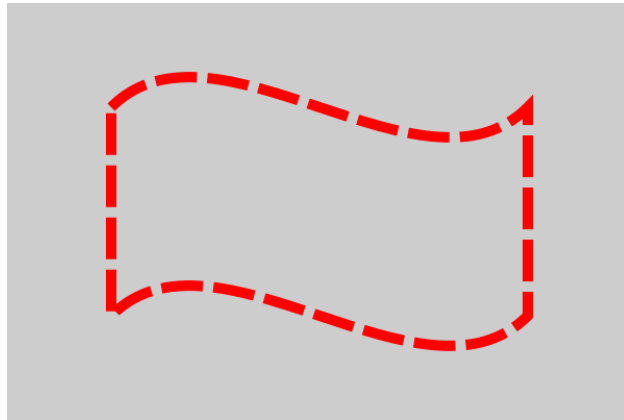
The dash pattern is specified by a list of values that control a repeating sequence of on/off lengths. The lengths are measured in the same units as the line width.

1. The top line is drawn with the pattern `[20]`. The list is repeated indefinitely to give a sequence 20, 20, 20, 20 ... This results in pattern of a 20 unit dash, 20 unit gap, 20 unit dash, 20 unit gap and so on. For brevity we will call this 20 on, 20 off, 20 on, 20 off ... It is an evenly spaced line of dashes and gaps.
2. The next line has the pattern `[20, 10]`. This gives a pattern 20 on, 10 off, 20 on, 10 off ... This is similar to the previous pattern but the gaps are shorter than the dashes.
3. This has a pattern `[20, 5, 5, 5]`. This gives a 20 on, 5 off, 5 on, 5 off ... It is a more complex pattern. The gaps are always 5, but the dashes alternate between 20 and 5.
4. This has a pattern `[5, 5, 10]`. This has an odd number of elements, but it cycles through them in the same way. It gives a pattern 5 on, 5 off, 10 on, then continues with 5 off, 5 on, 10 off, then repeats. This creates a pattern of length 6. It behaves exactly the same as `[5, 5, 10, 5, 5, 10]`.
5. The fifth example uses round line caps rather than butt. This gives a different effect because each dash has rounded ends. One thing to notice is that the semi-circular ends are added onto the original dash length. Although the pattern is 10 on, 20 off, the line appears to have dashes that are longer than the gaps. This is because each dash has an effective length of 20 (since it has a 5 unit semi-circle added to each end). Each gap has an effective length of 10 because the dashes have stolen some of its space. The total length of each dash-gap pair is still 30.
6. The final example illustrates a nice trick. The pattern is `[0, 20]`. The dash length is 0, but this zero length line still has a semicircle added to the start and end, creating a circular dot.

Of course, dashes don't just apply to straight lines. We can apply a dashed lines to the outline of our



Bezier shape from before:



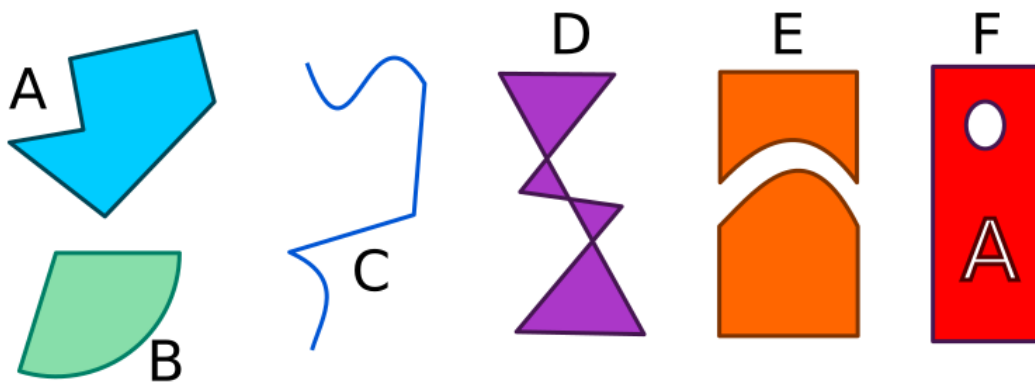
This is exactly the same code as the original Bezier example, but with a dash of `[40, 10]` applied. Notice that the dashes follow the curves, and even flow around the corners.

# 4 Paths and complex shapes

In the previous chapter *Basic drawing operations* we looked at ways to draw simple shapes. In this chapter we will extend this to look at more complex shapes, with practical examples. We will also take a closer look at paths and sub-paths.

## 4.1 Paths

A *path* is the most general type of shape. It is made up of one or more edges, than can consist of straight lines, arcs, or curves (specifically Bezier curves).



For example:

- Path A is created from 6 straight lines.
- Path B consists of 2 straight lines and a circular arc.
- Path C consists of 2 straight lines and 2 Bezier curves.

Each of the paths in the image has its own characteristics, to illustrate the variety of different forms a path can take:

- A and B are simple closed paths, that each enclose a single area. A is a polygon, B is a sector of a circle.
- C is an open path, it is just a set of lines that don't enclose an area. An open path has two end points that are not joined together.
- D is a self-intersecting path. It is a polygon just like A, but some of its sides cross over, so it encloses several different areas (4 triangular areas in this case).

- E is a disjointed path. It consists of 2 separate closed shapes (each shape is actually a sub-path, which we will explain in the next section).
- F is also a disjointed path, consisting of 3 separate closed shapes, but this time the two small shapes (the ellipse and the letter A) actually cut holes in the main rectangular shape. If you placed this shape on top of an image background, you would be able to see the image through the holes.

As example F shows, paths can be text-shaped (the red shape has a text-shaped hole).

Paths are created implicitly, by calling drawing operations on the Pycairo context. But it is also possible to store the path you have created, as a `Path` object. This can be very useful sometimes as we will see.

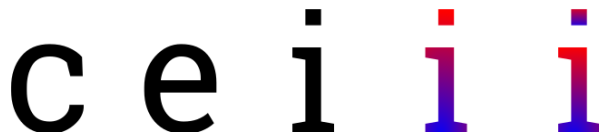
## 4.2 Sub-paths

Any path is made up of one or more *sub-paths*. A sub-path is a set of lines or curves joined end to end – each line starts where the previous one ended. A sub-path therefore creates either a single closed shape, or a single set of sequentially joined lines.

In the picture above, each of the four shapes on the left is an example of a path containing a single sub-path. The two shapes on the right are each examples of single paths that are made up of more than one sub-path:

- The orange shape labelled E is made up of two sub-paths (each of the two closed shapes is a separate sub-path).
- The red shape labelled F is made up of three sub-paths (the rectangle itself is one sub-path, the circular hole is another, the A-shaped hole is another).

Letter shapes are a good, practical example of using sub-paths.



Each of these letters is drawn as a separate path. The letter ‘c’ is a simple closed path. The letter ‘e’ consists of a closed path but it also has a hole in it. This is formed as a sub-path, similar to the example of red shape F, above. Paths with holes aren’t just some fancy effect you might never use, the page you are reading contains hundreds of examples.

The letter ‘i’ is interesting. It consists of two separate shapes (the main character and the dot above it), but of course they are part of the same letter shape. You wouldn’t usually want to show one without the other, and you would always want them to be in the same position relative to each

other – if you put the dot underneath the main character, for example, it wouldn't be a letter 'i' any more! So it makes perfect sense to have those two shapes as two sub-paths of the same path.

The second letter 'i' shows another advantage of sub-paths. We have filled this letter with a gradient (the colour changes from blue to red, vertically). Since the two shapes are both sub-paths of the same path, the gradient is applied across the entire letter, from the base right to the top of the dot.

The third letter 'i' shows how the gradient might look if the two parts of the letter were drawn as completely separate paths. The dot now has its own blue to red gradient, which is probably not the effect you would want.

When we look at text in a later chapter, we will see that often a whole word (or sentence, or paragraph) of text is drawn as a single path with lots of sub-paths (one or more for each letter).

## 4.2.1 Creating sub-paths

The most common way of creating a new sub-path is the `move_to` function. Here is some code that creates a path consisting of 3 sub-paths:

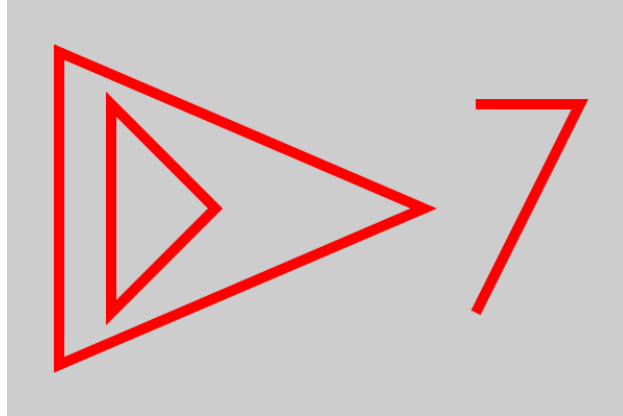
```
1  #Sub-path 1
2  ctx.move_to(50, 50)
3  ctx.line_to(400, 200)
4  ctx.line_to(50, 350)
5  ctx.close_path()
6
7  #Sub-path 2
8  ctx.move_to(450, 100)
9  ctx.line_to(550, 100)
10 ctx.line_to(450, 300)
11
12 #Sub-path 3
13 ctx.move_to(100, 100)
14 ctx.line_to(200, 200)
15 ctx.line_to(100, 300)
16 ctx.close_path()
17
18 ctx.set_source_rgb(1, 0, 0)
19 ctx.set_line_width(10)
20 ctx.stroke()
```

Sub-path 1 is started by the first `move_to`. It is a closed triangle with corners at (50, 50), (400, 200) and (50, 300). The call to `close_path` at the end of the definition closes the sub-path (ie it joins the final point to the initial point).

Sub-path 2 is started by the second `move_to`. It is an open shape with two sides. There is no `close_path` because the shape is not closed.

Sub-path 3 is started by the third `move_to`, and creates another closed triangle with corners at (100, 100), (200, 200) and (100, 300).

The final call to `stroke` draws the entire path (containing 3 sub-paths) and clears the path afterwards. Here is the image created:



You can also create a new sub-path using the context function `new_sub_path`. This is similar to `move_to`, but it doesn't set the current point. It is mainly used with the `arc` function as described later in this chapter.

The final way to create a new sub-path is the `rectangle` function. This creates a closed rectangular sub-path, see later in this chapter.

## 4.3 Lines

We know how to draw a line between two points (x1, y1) and (x2, y2), like this:

```
1 ctx.move_to(x1, y1)
2 ctx.line_to(x2, y2)
```

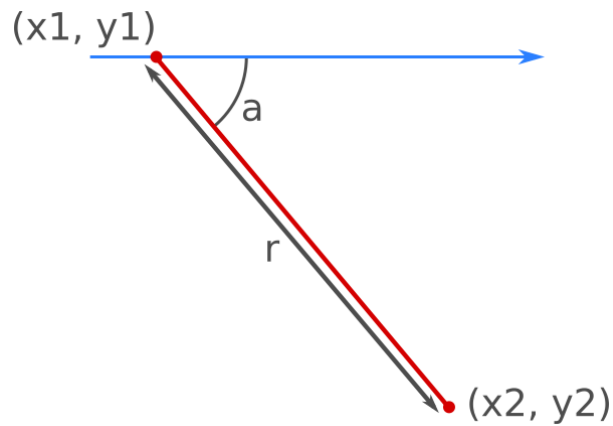
Now we will look at a couple of other examples.

### 4.3.1 Drawing a line of given length and angle

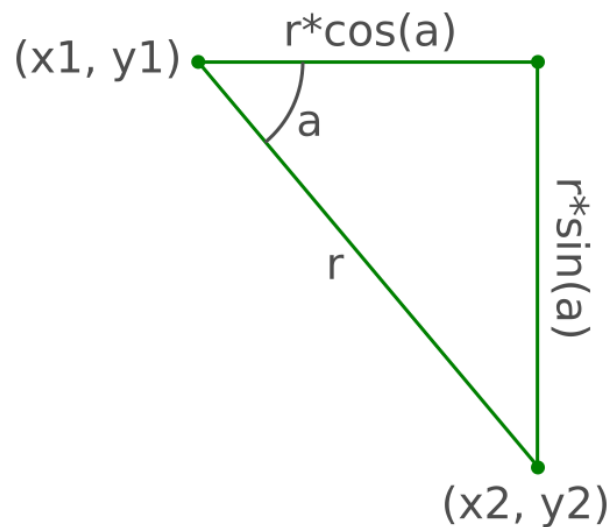
In this case we will see how to draw a line that:

- Starts from the point (x1, y1).
- Is at an angle  $a$  from the x-axis.

- Has length  $r$ .



To draw this line, we need to find the position of the second point  $(x_2, y_2)$ . We can use trigonometry to do this. We use the line to form a right-angled triangle, with the two short sides aligned with the x-axis and y-axis:



From this diagram we can see that:

- 1  $x_2 = x_1 + r \cdot \mathbf{math.cos(a)}$
- 2  $y_2 = y_1 + r \cdot \mathbf{math.sin(a)}$

So, the code to draw the line of a given length and angle is:

```

1 ctx.move_to(x1, y1)
2 ctx.line_to(x1 + r*math.cos(a), y1 + r*math.sin(a))

```

### 4.3.2 Relative drawing functions

In the example above, the position of  $(x_2, y_2)$  is calculated as an offset from the current point  $(x_1, y_1)$ . Its relative position is calculated.

The function `rel_line_to` allows us to specify the new point relative to the current point. We can simplify our drawing code like this:

```

1 ctx.move_to(x1, y1)
2 ctx.rel_line_to(r*math.cos(a), r*math.sin(a))

```

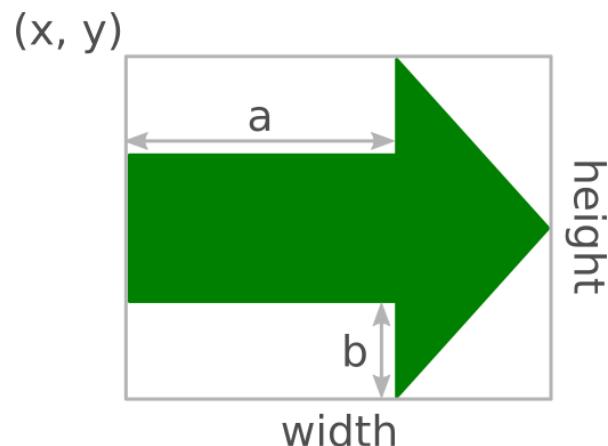
There is also a `rel_move_to` function and a `rel_curve_to` function, that operate in a similar way.

## 4.4 Polygons

In this section we will look at a slightly more complex polygons – a useful arrow shape. This example will illustrate a few techniques that can be used generally to design polygons.

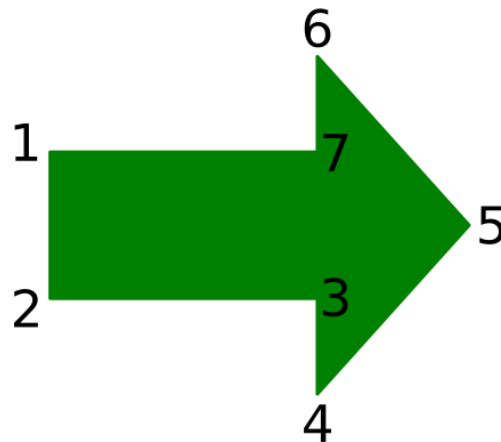
### 4.4.1 A simple arrow

Here we will see how to draw this arrow:



The size and position of the arrow is defined by its bounding box  $(x, y, \text{width}, \text{height})$ . The shape of the arrow can be adjusted by controlling the tail length  $a$  and inset  $b$ .

The shape has 7 vertices, which we will number:



We can calculate the position of each vertex:

1.  $(x, y + b)$
2.  $(x, y + \text{height} - b)$
3.  $(x + a, y + \text{height} - b)$

And so on. To draw the arrow, we just join the points to make a polygon. Here is a function that draws an arrow:

```

1 def arrow(ctx, x, y, width, height, a, b):
2     ctx.move_to(x, y + b)
3     ctx.line_to(x, y + height - b)
4     ctx.line_to(x + a, y + height - b)
5     ctx.line_to(x + a, y + height)
6     ctx.line_to(x + width, y + height/2)
7     ctx.line_to(x + a, y)
8     ctx.line_to(x + a, y + b)
9     ctx.close_path()

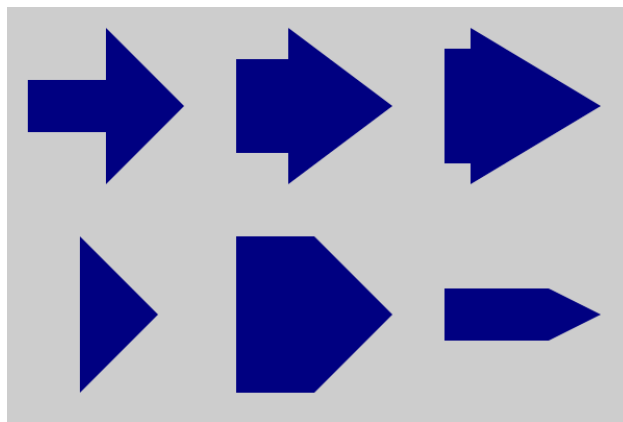
```

This code is quite versatile, we can create different arrow shapes by varying the size of the bounding box (width and height) and changing  $a$  and  $b$ . Here are some examples:



```
1 ctx.set_source_rgb(0, 0, 0.5)
2 arrow(ctx, 20, 20, 150, 150, 75, 50)
3 ctx.fill()
4 arrow(ctx, 220, 20, 150, 150, 50, 30)
5 ctx.fill()
6 arrow(ctx, 420, 20, 150, 150, 25, 20)
7 ctx.fill()
8 arrow(ctx, 70, 220, 75, 150, 0, 50)
9 ctx.fill()
10 arrow(ctx, 220, 220, 150, 150, 75, 0)
11 ctx.fill()
12 arrow(ctx, 420, 270, 150, 50, 100, 0)
13 ctx.fill()
```

This is the image the code creates:



## 4.5 Arcs

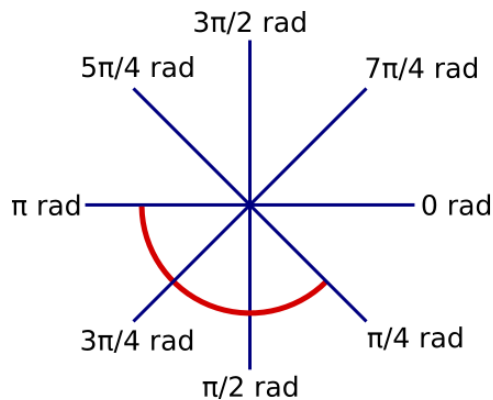
We took a quick look at arcs in an earlier chapter. This time we will look more closely at how angles are interpreted and how arcs connect to other lines and curves.

### 4.5.1 Arc angle rules

The arc function looks like this:

```
1 ctx.arc(x, y, r, a1, a2)
```

It draws an arc of a circle, centre  $(x, y)$  and radius  $r$ . The arc starts at angle  $a1$  and ends at angle  $a2$ . Here is an example:



The red arc was drawn with the following call:

```
1 ctx.arc(x, y, r, math.pi/4, math.pi)
```

There are several things to notice this:

- The angle 0 points horizontally to the left, that is in the direction of the x-axis.
- Angles increase as you move clockwise.
- The arc is drawn from the first angle to the second angle in the direction of increasing angles (ie clockwise).



These points are true of the default coordinate system that we are using in this chapter. If you transform the coordinates (as we will see later in the book) these directions can change. We will assume the default coordinates for the rest of this chapter.

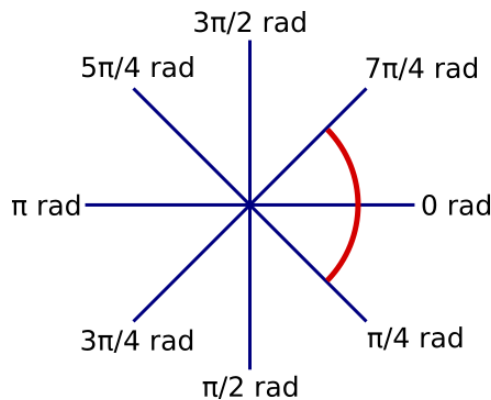
The arc is drawn as follows:

- The start of the arc  $a_1$ , i.e.  $\pi/4$ .
- The length of the arc is  $a_2 - a_1$ , i.e.  $(\pi - \pi/4)$  which is  $3\pi/4$ .
- Therefore, the arc starts at  $\pi/4$  and covers an angle of  $3\pi/4$ .

Here is another example:

```
1 ctx.arc(x, y, r, 7*math.pi/4, math.pi/4)
```

In this example,  $a_2$  is less than  $a_1$ . In that case Pycairo adds  $2\pi$  to  $a_2$ , repeatedly, until the result is greater than  $a_1$ . In this case we only need to add  $2\pi$  once, giving  $9\pi/4$ . This creates the following arc:



- The start of the arc  $a_1$ , i.e.  $7\pi/4$ .
- The length of the arc is  $a_2 - a_1$ , i.e.  $(\pi/4 - 7\pi/4)$  which is  $\pi/2$ .
- Therefore, the arc starts at  $7\pi/4$  and covers an angle of  $\pi/2$ .

Here are some more examples:

```
1 ctx.arc(x, y, r, 0, 0)           # 1
2 ctx.arc(x, y, r, 0, 2*math.pi) # 2
3 ctx.arc(x, y, r, 0, 3*math.pi) # 3
```

Case #1 will draw an arc of length 0. The rule above says that we must add  $2\pi$  to  $a_2$  if  $a_2$  is less than  $a_1$ . But in this case the two angles are equal, so  $a_2$  is not less than  $a_1$ . We don't need to add  $2\pi$ .

Case #2 has an arc angle of  $2\pi$ , which is a full circle. This code draws a complete circle.

Case #3 has an arc angle of  $3\pi$ , which is a one and a half full circles. This code draws a complete circle, but the current point is left at the position determined by the actual final angle of  $3\pi$ .

## 4.5.2 Arcs and the current point

When we use the `line_to` function, we only specify an end point. The function draws a line from the current point to the supplied end point.

`arc` is slightly different. By specifying the start and end angles, it implicitly defines the start and end points of the curve it draws. So how does the current point fit in?

There are two cases:

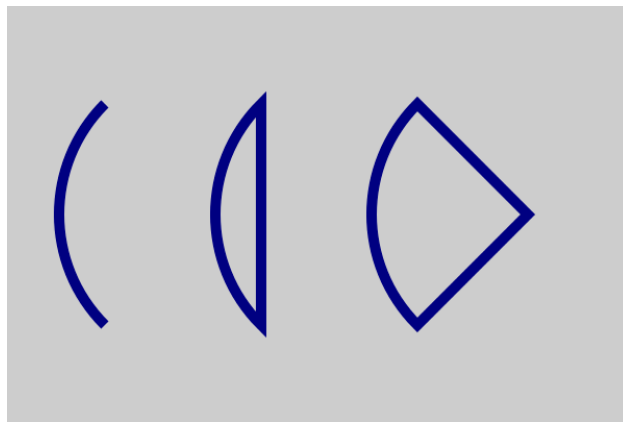
- If you create an arc when there is no current point defined, the arc starts at  $a_1$  as in the examples so far.

- If you create an arc when there is already a current point defined, a straight line is created from the current point to  $a_1$ , then the arc is drawn as normal.

In both cases, the current point is set to the end of the arc afterwards. We will see some examples of this, next.

### 4.5.3 Sectors and segments

The image below shows an arc, a segment and a sector. To illustrate how arcs interact with the current point, all three shapes are added as sub-paths within a single path.



Here is the drawing code:

```

1  ctx.set_line_width(10)
2  ctx.set_source_rgb(0, 0, 0.5)
3
4  # Arc
5  ctx.arc(200, 200, 150, 3*math.pi/4, 5*math.pi/4)
6
7  # Segment
8  ctx.new_sub_path()
9  ctx.arc(350, 200, 150, 3*math.pi/4, 5*math.pi/4)
10 ctx.close_path()
11
12 # Sector
13 ctx.move_to(500, 200)
14 ctx.arc(500, 200, 150, 3*math.pi/4, 5*math.pi/4)
15 ctx.close_path()
16
17 ctx.stroke()
```

The first (left hand) figure is just an arc. Since we are at the start of a new path, there is no current point, so we can just draw the arc as a standalone shape.

The second (centre) figure is a segment – that is, an arc with its ends joined. However, at this point we already have a current point defined – it is set to the end of the previous arc. If we just draw another arc, we will get an unwanted line from the old arc to the new one.

We can avoid this by calling `new_sub_path`. This function starts a new sub-path but without defining a current point. It is like calling `move_to` except it leaves the current point undefined.

We then draw the arc, followed by a call to `close_path`. The call to `close_path` draws a line from the current point (the end of the arc) right back to the first point in the current sub-path (the start of the arc). This creates the segment shape.

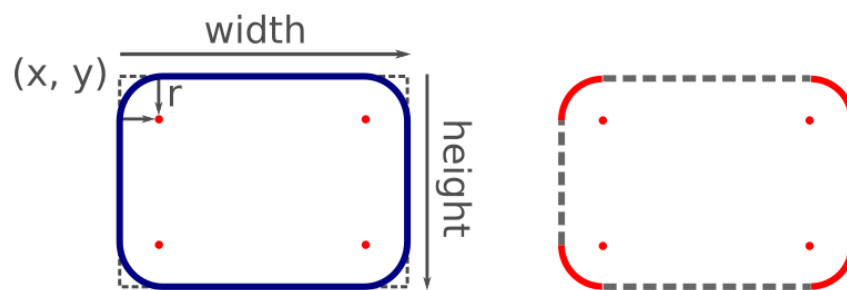
The final shape, on the right, is a sector – a pie wedge. We draw this shape like this:

- `move_to` to the centre of the arc. This starts a new sub-path with a defined current point.
- Add the arc. This automatically adds a line from the current point at the centre of the arc to the start point of the arc, and then adds the arc itself.
- Call `close_path`. This adds another line from the end of the arc to the start of the sub-path (ie the centre of the arc).

Notice that we have not explicitly drawn either of the straight lines! In the next example we will draw a `roundrect` shape without drawing any straight lines at all.

## 4.5.4 Roundrect

A `roundrect` is a rectangle with rounded corners:

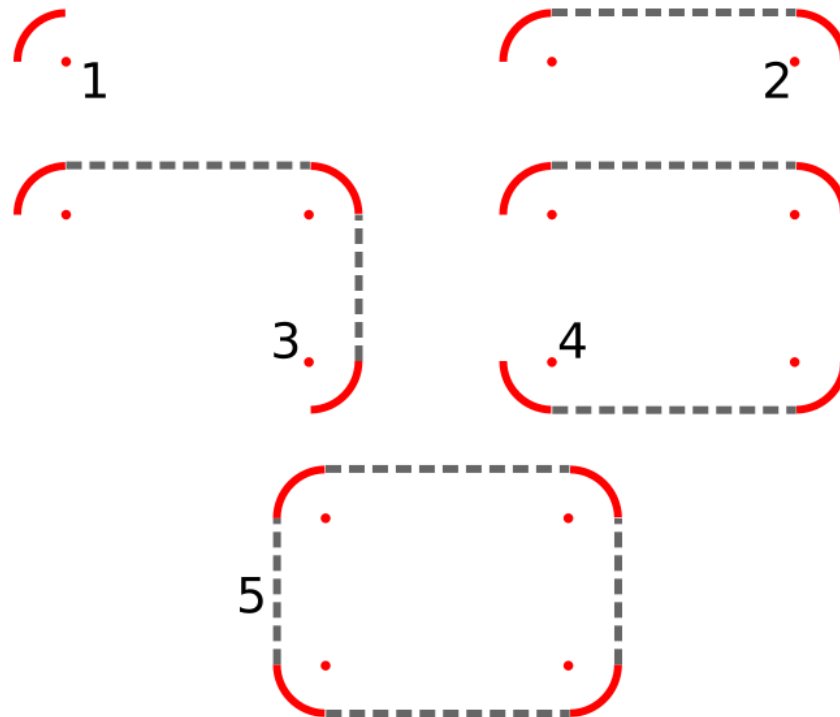


It is defined by its position  $(x, y)$ , width, height, and the radius  $r$  of the rounded corners. As the right-hand side of the diagram shows, a `roundrect` is made up of four quarter circles of radius  $r$ , with four straight lines between them.

The most important points for drawing a `roundrect` are the centres of the corner circles. These are inset from the corners of the enclosing rectangle by an amount  $r$ . Starting from the top left corner and working clockwise, these centre points are:

- $(x + r, y + r)$
- $(x + \text{width} - r, y + r)$
- $(x + \text{width} - r, y + \text{height} - r)$
- $(x + r, y + \text{height} - r)$

The diagram below shows how we create the roundrect. The red solid lines show the arcs that we explicitly draw, the grey dashed lines show the connecting lines that Pycairo draws automatically:



1. We draw the first quarter circle arc.
2. We draw the second quarter circle arc. Pycairo automatically draws a line from the end of the previous arc to start of the new arc.
3. We draw the third quarter circle arc, and again Pycairo draws a line from the end of the previous arc to the start of the new one.
4. We draw the fourth quarter circle arc, and again Pycairo draws the extra line.
5. Finally we close the path, so Pycairo adds a line from the end of the last arc to the start of the first arc, completing the shape.

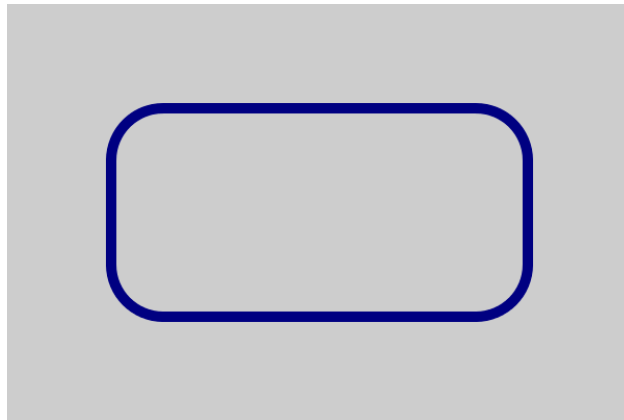
Here is the code to do this:

```

1  def roundrect(ctx, x, y, width, height, r):
2      ctx.arc(x+r, y+r, r, math.pi, 3*math.pi/2)
3      ctx.arc(x+width-r, y+r, r, 3*math.pi/2, 0)
4      ctx.arc(x+width-r, y+height-r, r, 0, math.pi/2)
5      ctx.arc(x+r, y+height-r, r, math.pi/2, math.pi)
6      ctx.close_path()
7
8  ctx.set_line_width(10)
9  ctx.set_source_rgb(0, 0, 0.5)
10 roundrect(ctx, 100, 100, 400, 200, 50)
11 ctx.stroke()

```

Here is the final result:



### 4.5.5 arc-negative

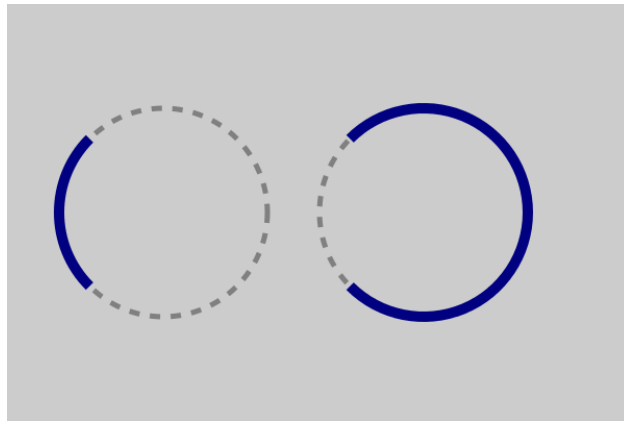
The `arc_negative` function is exactly the same as the `arc` function, except that the arc goes in the opposite direction. This code illustrates the difference:

```

1  ctx.set_line_width(10)
2  ctx.set_source_rgb(0, 0, 0.5)
3  ctx.arc(150, 200, 100, 3*math.pi/4, 5*math.pi/4)
4  ctx.stroke()
5  ctx.arc_negative(400, 200, 100, 3*math.pi/4, 5*math.pi/4)
6  ctx.stroke()

```

The `arc` function draws an arc from angle  $3\pi/4$  to angle  $5\pi/4$  in the direction of increasing angles (clockwise by default). The `arc_negative` function draws an arc from angle  $3\pi/4$  to angle  $5\pi/4$  in the opposite direction. In the image, `arc` is on the left and `arc_negative` is on the right (the dashed circle illustrates the whole circle in each case):

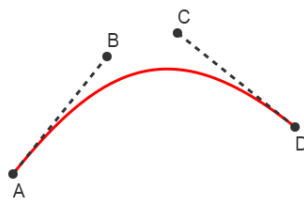


You can do the same thing by swapping the start and end angles in the standard `arc` function, but `arc_negative` can be more intuitive in some cases.

## 4.6 Bezier curves

A *Bezier curve* is a commonly used curve in vector graphics. It is versatile, efficient to work with, and has many useful properties.

A *cubic* Bezier curve is the most often used type, and is actually the only one supported by Pycairo. It is defined by four points A, B, C, D:



The points A and D are called the *anchors*, and are always located at the two ends of the curve. The points B and C are called the *handles*, and control the basic shape of the curve. The curve will generally follow the form of the open shape ABCD.

Pycairo defines a Bezier curve with the function `curve_to`:

```
1 ctx.move_to(ax, ay)
2 ctx.curve_to(bx, by, cx, cy, dx, dy)
3 ctx.set_source_rgb(1, 0, 0)
4 ctx.set_line_width(2)
5 ctx.stroke()
```

In the code, point A is represented by coordinates `(ax, ay)`, point B by `(bx, by)` etc.



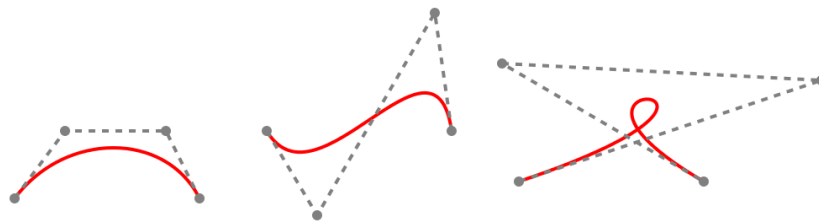
The curve uses `move_to` to set the current point as the point A. The `curve_to` function treats the current point as point A, and accepts the coordinates of points B, C, D as parameters.

### 4.6.1 Common forms

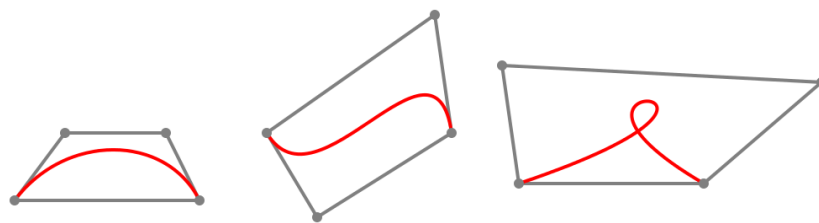
If you keep the end points A and D fixed, you can create a variety of different shapes by moving the handles B and C around. There are 3 basic forms:

- A simple curve.
- An S shaped curve.
- A looped curve.

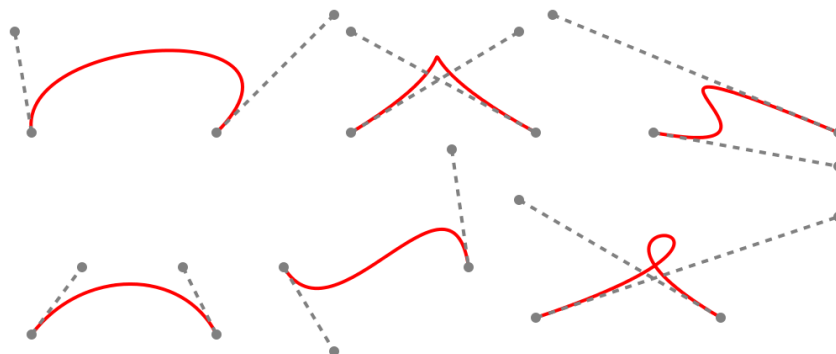
These are illustrated here.:



The curve always fits entirely within the *convex hull* of the points A, B, C, D:



Here is a larger selection of possible shapes:



The best way to explore Bezier curves is probably to use a vector graphics editor. Inkscape is an excellent, free product that runs on Windows, Linux and Mac. You can experiment with shapes and read off the positions of the nodes when you have a shape you like.

## 4.6.2 Splines

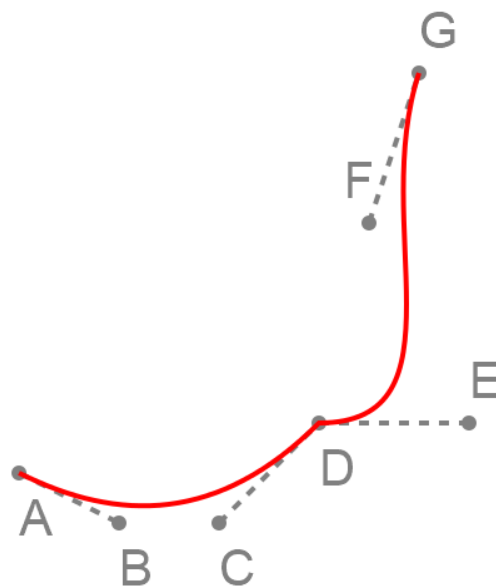
Sometimes you might want to draw a more complex curved shape. For example in a 2D computer game you might have characters, spaceships, clouds, etc. These can be drawn by joining several Bezier curves together. A composite curve, constructed from more than one base curve joined end to end, is called a spline. Bezier curves are an excellent choice for making splines. Not only are they easily adjustable to match a shape locally, but they can also be joined smoothly.

It is quite easy to join two or more Bezier curves. We just need to define the curves so that they have a shared anchor point.

The diagram below shows two Bezier curves that are joined. The first curve has anchor points A and D, with handles B and C. The second curve has anchor points D and G, with handles E and F. Since the curves both have point D as an anchor, they are joined at that point.

The code to draw two joined Bezier curves looks like this:

```
1 ctx.move_to(ax, ay)
2 ctx.curve_to(bx, by, cx, cy, dx, dy)
3 ctx.curve_to(ex, ey, fx, fy, gx, gy)
```



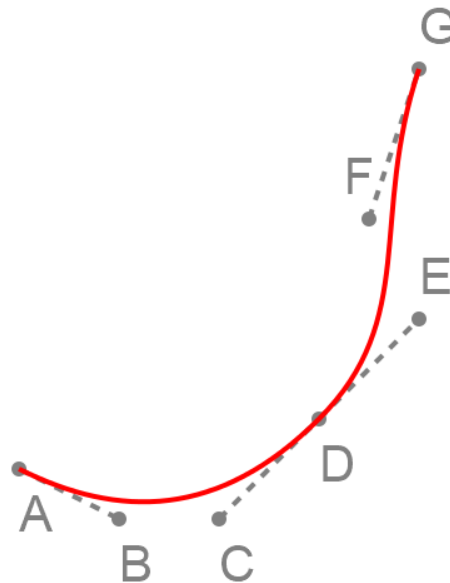
Here is how it works:

- `move_to` sets the current point to point A.
- `curve_to` draws a curve from the current point A to the point D, using B and C as handles.
- After executing `curve_to` the current point is set to D
- The second `curve_to` draws a curve from the current point D to the point G, using E and F as handles.

### 4.6.3 Smooth curves

In the diagram above, you will notice that the two Bezier curves form a corner where they connect. There is nothing wrong with that, it is sometimes what you want to create a particular shape.

Other times you might want two Bezier curves to join more smoothly, so it looks like a single curve, as shown below:



It is fairly easy to do this. The slope of a Bezier curve at its end point is equal to the slope of the line from the handle. That is:

- For the first curve, the slope of the curve at point D is the same as the slope of the line between C and D.
- For the second curve, the slope of the curve at point D is the same as the slope of the line between D and E.

This means that if the points C, D and E are all in a straight line (as they are in the example), the two curves will have the same slope where they meet, which avoids a corner.

In other words, to obtain a smooth join, make sure the handles of the two curves line up.

Another thing to know is that the second derivative of the curve at its end point is determined by the length of the line to the handle. The second derivative means the rate of change of the slope, or very loosely speaking how curved the Bezier curve is near its end point. If the two curves have the same slope and curvature, the join will be even more smooth.

So, to get the smoothest result when joining two curves, ensure that:

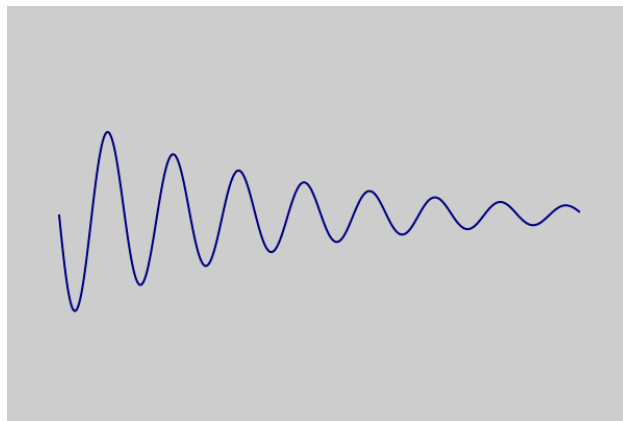
- The handles C and E, and the anchor D are in a straight line.
- The distance from C to D is the same as the distance from D to E .

## 4.7 Function curves

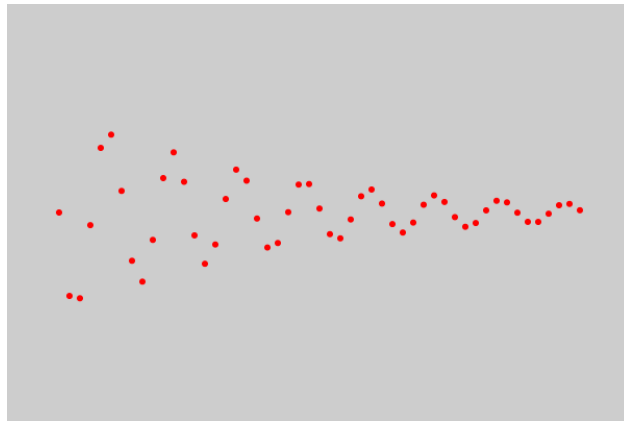
Sometimes you might want to draw a curve based on a mathematical function. Here is an example of a function  $y = f(x)$ :

```
1 y = math.sin(10*x)*math.exp(-x/2)
```

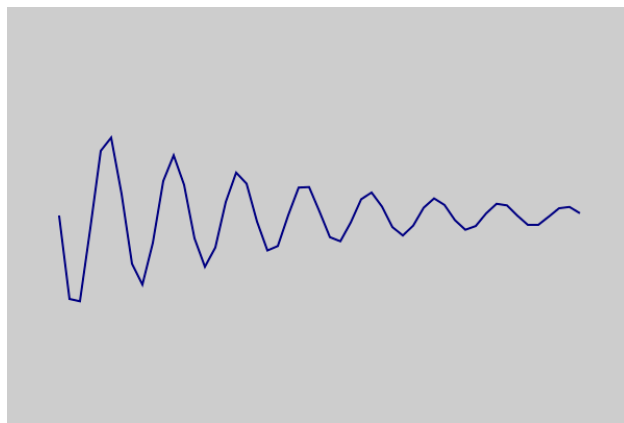
This function is a *decaying sine wave*. It represents simple damped oscillation - for example, the sound created if you pluck a guitar string, that starts off loud and gradually fades away. Don't worry too much about the details of the function, we are mainly using it because it looks quite nice. This is a graph of the function created using Pycairo:



How is that done? Well, it is actually quite simple. We choose a series of value of  $x$ , and calculate  $f(x)$  for each value. This gives us a set of points that lie on the curve:



We then join these points with straight lines. In other words, we create a polyline from all the points.



Now this looks pretty lumpy at the moment, you can clearly see the individual lines. That is deliberate, to show what is going on. The trick is to calculate more points that are closer together, then they look like a smooth curve.



This is actually very similar to what Pycairo does when it draws a Bezier curve or an arc. It splits the curve into a series of very small straight line segments, and draws those. This is called *flattening* the curve. It looks like a smooth curve but it is actually just straight lines.

Now we can move on to the actual code that creates the function curve above. Here is how we calculate the points:

```

1 x = 0
2 points = []
3 while x < 5:
4     y = math.sin(10*x)*math.exp(-x/2)
5     points.append((x*100 + 50, y*100 + 200))
6     x += 0.01

```

The while loop iterates over values of  $x$  from 0.0 to (almost) 5.0 in steps of 0.01. The step value is very important, it determines the gap between the calculated points, which controls how smooth the curve is. You need to experiment a bit, if the gap is too big the curve won't be smooth, but if it is too small you will be calculating more points than you really need to.

For each  $x$  value, we calculate a  $y$  value using our function (the decaying sine wave function).

The result to of this is a set of  $(x, y)$  values that we want to plot. But there is a slight problem. Our  $x$  values are in the range 0.0 to 5.0. Our  $y$  values are in the range -1.0 to +1.0. If we plot those points on a canvas that is 600 units by 400 units, we will end up with a tiny graph stuck in the top corner!

We need to scale and translate our  $x$  and  $y$  values to create a set of points that have sensible canvas locations. Here is what we do:

- For  $x$  values we multiply by 100 and add 50. So  $x$  values in the range 0.0 to 5.0 create canvas values in the range 50 to 550 - ideal for a canvas that is 600 units wide.
- For  $y$  values we multiply by 100 and add 200. So  $y$  values in the range -1.0 to +1.0 create canvas values in the range 100 to 300 - also ideal for a canvas that is 400 units high.

We add these scaled points to the `points` list, as  $(x, y)$  tuples.

Plotting the points is now quite easy:

```

1 ctx.move_to(*points[0])
2 for p in points[1:]:
3     ctx.line_to(*p)
4
5 ctx.set_line_width(2)
6 ctx.set_source_rgb(0, 0, 0.5)
7 ctx.stroke()

```

We `move_to` to the location of `points[0]`, and then call `line_to` on the rest of the points in the list to create a polyline. We then stroke the polyline.

## 4.8 Rectangle

We met the `rectangle` function in the chapter *Basic drawing operations*. There isn't a lot more to say about it really, it draws a rectangle!

The call `rectangle(x, y, width, height)` is roughly equivalent to this code:

```
1 ctx.move_to(x, y)
2 ctx.line_to(x + width, y)
3 ctx.line_to(x + width, y + height)
4 ctx.line_to(x, y + height)
5 ctx.close_path()
```

It creates a new sub-path with `move_to` and closes the path when it has finished.

# 5 Computer colour

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 5.1 RGB colour

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 5.1.1 Why RGB?

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 5.2 Pycairo RGB colours

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 5.2.1 Primary colours

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 5.2.2 Secondary colours

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 5.2.3 All colours

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.



## 5.3 CSS named colours

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 5.4 Transparency

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 5.5 Transparency colour calculation

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 5.6 Transparent images

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 5.7 Greyscale images

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 5.8 Pixel colours

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 5.8.1 Other image formats

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

# 6 Transforms and state

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 6.1 User space and device space

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 6.2 Translation

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 6.3 Scaling

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 6.3.1 Scaling down

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 6.3.2 Unequal scaling

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 6.4 Rotation

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 6.5 Save and restore

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 6.5.1 Save and restore are stacked

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 6.6 Rotating about a point

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 6.7 Placing an ellipse

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 6.8 Correcting the effects of unequal scaling

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 6.9 Flipping

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 6.10 Current transform matrix

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

# 7 Working with text

Text is an important feature of vector graphics. Whether you are creating a diagram, chart, illustration, logo or whatever, there is a good chance you will need to include text at some point. This book contains mainly text, and you are reading it electronically, it will most likely be stored in a vector format such as PDF.

In this chapter we will look at how to handle text in Pycairo, including:

- Adding simple text labels to an image.
- Choosing the font and size of your text.
- Aligning text with other objects.
- Text effects such as outlined text, gradient fills and text shaped holes.
- Making text follow a curve or path.
- The basic concepts of more advanced typography.

## 7.1 Text is just shapes

At its most basic level, text is just a collection of shapes. Take a look at the huge text below:

The word "text" is displayed in a large, bold, black serif font. The letters are thick and have a classic, slightly ornate design with prominent serifs. The word is centered horizontally and takes up a significant portion of the page width.

Each letter is just a shape, created from straight lines and Bezier curves. And ultimately, Pycairo will draw each letter as a shape.

## 7.2 How Pycairo handles text

Of course, text isn't really just shapes. Those shapes have meaning - they are letters that combine to make words and sentences.

Trying to add text to your file by drawing the shape of each character would be ridiculously difficult and tedious. Pycairo handles text at a higher level. You normally specify the text as a text string, then set the size and font that you require.

You will no doubt be familiar with fonts, if you have ever used any kind of office application. You select a font to control the typeface that is used to display or print text. But what is a font? In simple terms, it is a file that defines the shapes of every letter, number and symbol in some defined character set, for a particular typeface.

When you add text in Pycairo, you use a normal Python string of characters. When you go to create a PNG file, Pycairo looks up the shape of each character in the selected font file, then renders it to pixels in the usual way. The font file also contains information about the size of each character, and Pycairo uses this information to correctly position each character in the string.

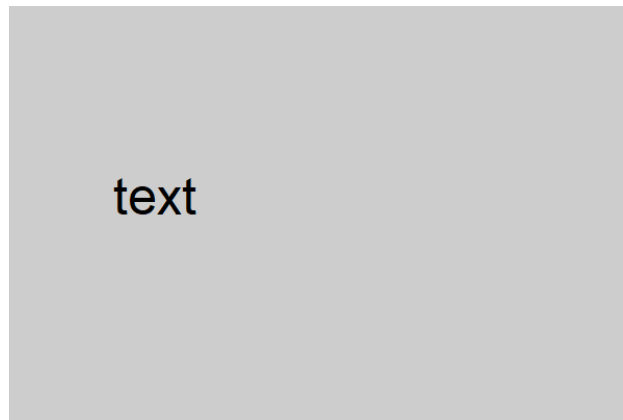
Here is the code to draw a simple text string:

```
1 ctx.select_font_face("Arial")
2 ctx.set_font_size(50)
3
4 ctx.set_source_rgb(0, 0, 0)
5 ctx.move_to(100, 200)
6 ctx.show_text("text")
```

Here are the basic stages:

1. Use `select_font_face` to choose the font our text. This is exactly the same as choosing a font in a word processor or spread sheet.
2. Use `set_font_size` to set the size. With the default scaling, this is the approximate character size in pixels (see later for an exact definition).
3. Set the colour to black in the usual way, with `set_sorce_rgb`.
4. Use `move_to` to set the current point. This will be the starting point if the text (the baseline position of the first character).
5. Finally `show_text` draws the characters.

Here is the result:



Notice that we only need to specify the one position (100, 200). This is the position of the first 't' in the string. The remaining characters 'e', 'x' and 't' are positioned automatically, using character metrics supplied by the font.

## 7.3 Fonts

Most computers have a selection of fonts installed on them. These can come from various sources:

- A default set of common fonts will have been supplied with the operating system.
- Many applications that support text formatting will come their own fonts, which are installed when the application is installed.
- The user may have manually installed additional fonts, either free fonts from various websites, or commercial fonts that have been purchased.

Each font is stored in its own file in a standard format (there are several different formats that are all well supported), and usually placed in a special system folder (eg *c:WindowsFonts* on Windows). This means that all your fonts from whatever source are potentially available to all your applications, and generally Pycairo can use them too.

Although there are many different fonts, but they mainly fit into five different families - serif, sans serif, monospace, cursive, and display. These are general categories, so some fonts might not fit easily into any of these families, but most do. This code displays an example of each type of font:

```
1  ctx.set_font_size(40)
2  ctx.set_source_rgb(0, 0, 0)
3
4  ctx.select_font_face("Times")
5  ctx.move_to(50, 100)
6  ctx.show_text("Serif - Times")
7
8  ctx.select_font_face("Arial")
9  ctx.move_to(50, 150)
10 ctx.show_text("Sans serif - Arial")
11
12 ctx.select_font_face("Courier New")
13 ctx.move_to(50, 200)
14 ctx.show_text("Monospace - Courier")
15
16 ctx.select_font_face("Freestyle Script")
17 ctx.move_to(50, 250)
18 ctx.show_text("Cursive - Freestyle Script")
19
20 ctx.select_font_face("Dayton")
21 ctx.move_to(50, 300)
22 ctx.show_text("Display - Dayton")
```

Here is the resulting image:



Your computer might not have a copy of each of the particular fonts in this example, but you should be able to choose a similar alternative if you want to run the code.

**Serif** fonts have marks (called serifs) that decorate the letter shapes. **Sans serif** fonts have a plainer design. This diagram shows the difference (the serifs on the 'S' are circles in red):



Serif  
San

There is a general belief that a well designed serif font is easier to read, perhaps because the letter shapes are more distinctive so the brain finds them easier and quicker to recognise. Traditionally, books, newspapers and magazines have used serif fonts for the main body text, and often sans serif fonts for headlines as they look neater. However, fonts are subject to fashion like everything else, and at the time of writing many websites use sans serif fonts throughout.

The fonts above, like most fonts, are *proportionally spaced fonts*. This means that different characters have different widths. For example, a letter 'm' is wider than a letter 'i'. With **Monospace** fonts, all characters have exactly the same width. This is shown below, the top line is a monospaced font, the bottom line is a proportionally spaced font.



m|i|i|i|m  
m|iii|m

Proportionally spaced fonts are generally preferred for most text, because the text looks more appealing, it is easier to read and you can get more text on the page without affecting readability. Monospace fonts are only really used where you have a specific reason to want characters that line up down the page in columns. The most common application is computer programming, where text is pretty much always shown in a monospace font.

While serif, sans serif and monospaced fonts have specific uses, the remaining font families exist solely for artistic effect. **Cursive** fonts are fonts that resemble handwriting. This can be in any style



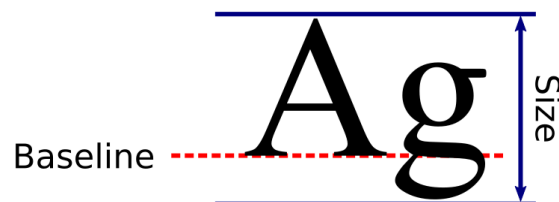
- formal or informal, joined up or separate characters. They can be used to indicate informality, or to show quoted text, or just to look nice.

**Display fonts** are basically everything else. Fonts that are unusual or designed to stand out or grab your attention. Heavily stylised fonts, novelty fonts, etc. They are often used on product packaging, SALE signs in shops, company logos and so on. They vary in readability, but you probably wouldn't want to read a novel printed with a display font.

## 7.4 Font size

The `set_font_size` function sets the size of the font (and therefore the size of any text you create with that font). But what exactly does the size value mean?

Font size is measured in user units - the same units you use when setting the line width. As a rule of thumb, the font size indicates the height of the text, from the top of the tallest alphabetic character such as a capital 'A', right down to the bottom of the tail of a character such as 'g'. Here is an example:



As you can see, the size includes a small extra margin above and below. Most fonts tend to do this.

If this seems a little vague, there is a good reason. There are no fixed rules about how a particular font interprets the size value, it is largely down to the font designer to decide what they provide when the user asks for a particular sized font. Most fonts behave as described here, but if you are using a font you haven't used before there may be some variation, so experiment first.

Also be aware that there is no guarantee that every single character in the font will fit within the specified height. For many fonts, all characters will be within the bounds, but some fonts might behave differently, for example some fonts have brackets that are larger than the normal alphabetic characters.

When using the default scaling (if you haven't applied any scaling via the `Context.scale` function or similar) one user unit is equal to one pixel, so for example a font size of 50 will produce text that is about 50 pixels high when you use Pycairo to create a PNG file.

### 7.4.1 Point size

Font size is usually measured in *points*. There are exactly 72 points in an inch, so a point is approximately a third of a mm (1 point is actually 0.352778 mm to 6 decimal places).

Thus, for example, text in an 18 point font would be expected to be 18/72 inches (ie 0.25 inches) tall, from the top of an ‘A’ to the tail of a ‘g’.

Point size mainly matters if you are intending to print the output, and you want the text on the physical printed page to be a specific size. You would need to take account of your printing resolution.

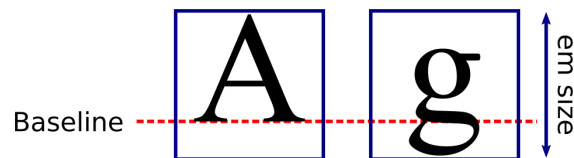
To give an example, suppose we wanted to print 18 point text, on a printer whose resolution was 144 pixels per inch. 18 point text should be 0.25 inches tall, which for the printer corresponds to 36 pixels, so you would want to create an image with text that is 36 pixels tall. At default scaling you would therefore need to set a font size of 36.

## 7.4.2 Em squares

So far we have dealt with fonts that contain characters from the Latin character set (that is, letters A-Z and a-z). We have defined the font height as the height of these characters. Any other characters, such as punctuation, numbers, symbols, are assumed to be proportional to the size of the letters.

What if we were dealing with a non-Latin character set, for example a Chinese font? It might not have any Latin characters at all, so how would we specify its size?

We can solve this by generalising the idea of the height, and instead defining a square that fits around most of the characters. It is called the em square, and the size of that square is the em size:



For Latin character fonts, the characters tend to be tall rather than wide, so the size of the em square is usually determined by the height of the characters. In other words, the em size is the same as the font height.

For some other writing systems where characters are wide rather than tall, the em square size is determined by the width of the characters in the font.

Remember that the em square is there to indicate the size of the font, it is not intended to be used for measuring the character spacing. In the example above, the em square is much wider than the ‘A’ or ‘g’ characters. That *doesn't* mean that you have to space put huge spaces between the characters!

## 7.5 Font style

In addition to the normal font style, text is often displayed in italic or bold styles. These styles are applied via optional parameters when you create the font.

Here is how to create italic fonts:

```
1 ctx.select_font_face('Serif', cairo.FontSlant.NORMAL)
2 ctx.select_font_face('Serif', cairo.FontSlant.ITALIC)
3 ctx.select_font_face('Serif', cairo.FontSlant.OBLIQUE)
```

The first example creates normal text using the default serif font. The font slant parameter is optional in this case, it defaults to NORMAL if it is left off.

The second and third examples create italic and oblique serif fonts.



Italic and oblique styles are both slanted forms of the basic font. In the oblique version of the font, the characters are identical to the normal font characters, simply slanted. In the italic version, the characters may slightly different to the normal font - often slightly more cursive or fancy. The difference is usually quite subtle. Some fonts don't have oblique or italic versions, they are created by automatically transforming the normal font. In that case, italic and oblique will usually be identical to each other. In any case, unless you are into advanced typography, it is safe to stick with the italic style.

Here is how to create bold fonts:

```
1 ctx.select_font_face('Serif', cairo.FontSlant.NORMAL, cairo.FontWeight.BOLD)
2 ctx.select_font_face('Serif', cairo.FontSlant.ITALIC, cairo.FontWeight.BOLD)
3 ctx.select_font_face('Serif', cairo.FontSlant.OBLIQUE, cairo.FontWeight.BOLD)
```

The first example creates a bold font.

The second and third examples create a font that is both bold and italic/oblique.

Here are examples of each case using the default serif font:



Normal  
*Italic*  
*Oblique*  
**Bold**  
***Bold Italic***  
***Bold Oblique***

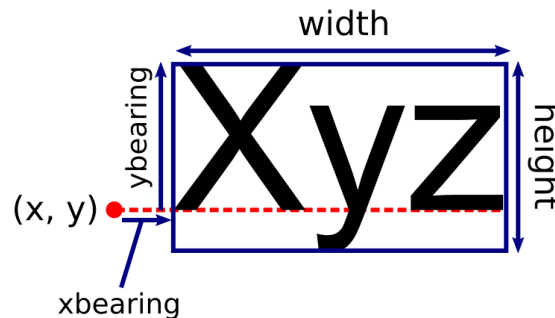
## 7.6 Text extents

The font size gives us control of the nominal size of the font. The *text extents* tell us the exact size of a particular text string when rendered with the current font. To be precise, they tell us the rectangle around the area that will be marked when we paint the text.

The Context function `text_extents` returns a Pycairo `TextExtents` object (which behaves much like a tuple). Here is an example:

```
1 xbearing, ybearing, width, height, dx, dy = ctx.text_extents('Xyz')
```

`text_extents` accepts a string. It calculates the extents of the string, and returns the `TextExtents` object, which we unpack into 6 variables. Here is an example:



In the diagram, the red dot represents the point  $(x, y)$  where the text will be positioned. This corresponds to the current point when `show_text` is called. For Latin characters, it will generally be on the text baseline.

The `width` and `height` give the size of the rectangle that tightly encloses the marked area when the text is painted. These numbers will always be non-negative. It is possible for the `width` and `height` to be zero - for example if the string contained only space characters, it wouldn't mark the page at all so the marked area would have zero `width` and `height`.

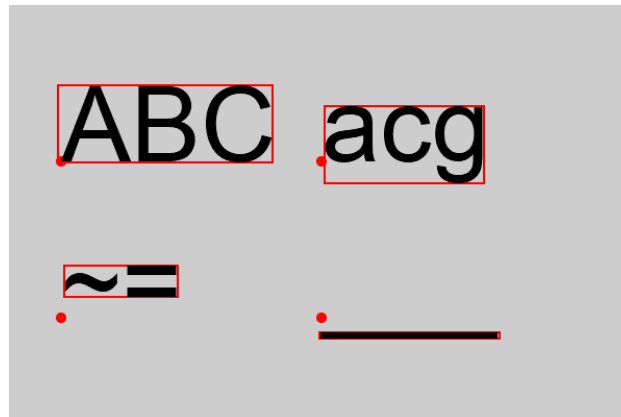
The `xbearing` is the distance from the point  $(x, y)$  to the left hand side of the text. The number is positive if, as shown, the text rectangle starts to the left of the point. It is negative if the text rectangle starts to the right of the point. For Latin characters, this offset is usually very small or zero, but can be positive or negative.

The `ybearing` is the distance from the point  $(x, y)$  to the top of the text rectangle. For Latin characters, the text will normally extend above the baseline, so the `ybearing` is *usually negative*. That isn't always the case, as we will see below.

`dx` and `dy` tell us where we should place the next text string if we want it to follow on directly from the current text. We will cover this later.

## 7.7 Text extent examples

In the image below, we show several text strings. For each string we show its text extents as a red outline, and the text position  $(x, y)$  as a red dot:



The text rectangle is calculated as follows:

- The top left corner is at location  $(x + \text{xbearing}, y + \text{ybearing})$ .
- The size of the rectangle is width by height.

Here are the text extents:

String	<code>xbearing</code>	<code>ybearing</code>	<code>width</code>	<code>height</code>	<code>dx</code>	<code>dy</code>
ABC	-3.0	-73.0	206.0	74.0	206.0	0.0
acg	3.0	-53.0	153.0	74.0	161.0	0.0
~ =	3.0	-50.0	109.0	30.0	116.0	0.0
—	-2.0	14.0	173.0	6.0	168.0	0.0

For the string ‘ABC’ the `xbearing` is -3, indicating that the string starts very close to the point  $(x, y)$ . This is generally true of Latin character strings. The `width` is simply the width of the string as it is displayed.

The `ybearing` is -73, indicating that the top of the text rectangle is well above the baseline. The `height` is 74. This means that the bottom of the rectangle is 1 below the text baseline. That is because the text contains capital letters with no tails, so the text doesn’t extend much below the baseline.

For the string ‘acg’, the `ybearing` is -53 but the `height` is 74. This indicates that the top of the text rectangle is well above the baseline, but the bottom of the rectangle is well below the baseline. That is due to the tail of the ‘g’ character.

The string ‘~ =’ contains two characters which hover above the baseline. The `ybearing` is -50 but the `height` is only 30, which means the entire text rectangle is up above the baseline, as you would expect.

The final string consists of three underscore characters ‘\_\_\_’. Underscores are unusual because they are placed below the baseline. In this case the `ybearing` is positive (because the top of the rectangle is below the baseline).

## 7.8 Text alignment

We can use the text extents to align text. AS an example we will look at how to align text to the left, right and centre. Here is a function that displays a string *left-aligned* to the point  $(x, y)$ :

```
1 def left_align(ctx, x, y, text):
2     ctx.move_to(x, y)
3     ctx.set_source_rgb(0, 0, 0)
4     ctx.show_text(text)
```

This is fairly standard, it sets the current point to  $(x, y)$ , sets the colour to black, then shows the text using the current font and size. The text will be automatically left-aligned because that is Pycairo’s default.

Here is the code to right-align text to the point  $(x, y)$ :

```
1 def right_align(ctx, x, y, text):
2     xbearing, ybearing, width, height, dx, dy = ctx.text_extents(text)
3     ctx.move_to(x - width, y)
4     ctx.set_source_rgb(0, 0, 0)
5     ctx.show_text(text)
```

In this case we make an extra call to `text_extents`. The only value we are really interested in is the `width` value.

We set the current point to  $(x - \text{width}, y)$ . This means that the starting point of any text string depends on its width. It will always start at  $x - \text{width}$ , which means that the string will always end at  $(x, y)$ .

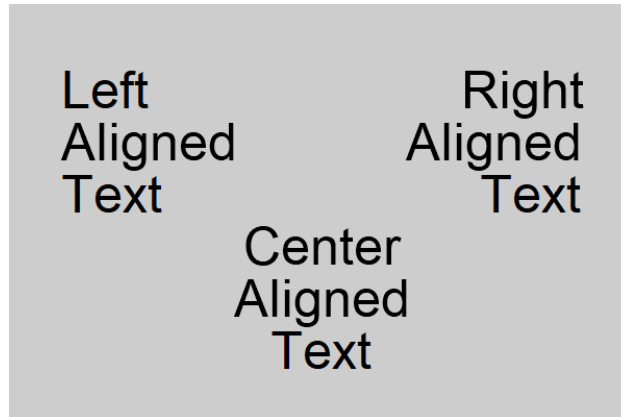


We only use the `width` value in this calculation. This assumes that `thexbearing` is zero (or at least, too small to care about). That is generally true of Latin characters, but might not be true of other writing systems.

We can also centre-align text by setting the current point to  $(x - \text{width}/2, y)$ :

```
1 def center_align(ctx, x, y, text):
2     xbearing, ybearing, width, height, dx, dy = ctx.text_extents(text)
3     ctx.move_to(x - width/2, y)
4     ctx.set_source_rgb(0, 0, 0)
5     ctx.show_text(text)
```

Here is an example of these three functions in action:



This is the code used to create the image above:

```
1 ctx.set_font_size(50)
2 ctx.select_font_face('Arial')
3
4 left_align(ctx, 50, 100, 'Left')
5 left_align(ctx, 50, 150, 'Aligned')
6 left_align(ctx, 50, 200, 'Text')
7
8 center_align(ctx, 300, 250, 'Center')
9 center_align(ctx, 300, 300, 'Aligned')
10 center_align(ctx, 300, 350, 'Text')
11
12 right_align(ctx, 550, 100, 'Right')
13 right_align(ctx, 550, 150, 'Aligned')
14 right_align(ctx, 550, 200, 'Text')
```

Notice that all the `left_align` calls use an `x` value of 50. That means that the left hand side of all three strings will be aligned at `x` position 50. All the `center_align` calls use an `x` value of 300. That means that the centre of all three strings will be aligned at `x` position 300. Similarly the `right_align` calls all align the right hand side of the text at 550.

# 8 Gradients and image fills

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 8.1 Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 8.2 SolidPattern

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 8.2.1 SolidPattern vs set\_source\_rgb

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 8.2.2 Creating a transparent SolidPattern

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 8.3 Linear gradient

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 8.3.1 Transparent stops

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.



## 8.4 Linear gradients at different angles

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 8.5 Adding more stops

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 8.5.1 Flat colours

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 8.5.2 Step changes

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 8.6 Extend options

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 8.7 Filling a stroke with a gradient

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 8.8 Filling text with a gradient

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 8.9 Radial gradients

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 8.10 Radial gradient with inner circle

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 8.11 Radial extend options

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 8.12 Loading an image into Pycairo

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 8.13 Using SurfacePattern with an image

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 8.13.1 Transforming the image

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 8.13.2 Tiling the image

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 8.14 SurfacePattern extend options

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 8.15 Using SurfacePattern with vectors

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

# 9 Clipping, masking and compositing

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 9.1 Clipping

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 9.1.1 Using clipping to create complex shapes

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 9.2 Calling clip multiple times

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 9.3 Resetting the clip region

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 9.4 Clipping functions

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 9.5 Masking

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 9.6 Using an image as a mask

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 9.7 Compositing

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 9.7.1 Compositing terms

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 9.7.2 Setting the operator

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 9.8 OVER operator

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 9.9 Changing the drawing order

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 9.10 Masking operations

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 9.10.1 Other masking modes

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 9.11 Artistic colour adjustments

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 9.12 Specific colour changes

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

# 10 Surfaces and output formats

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 10.1 Output formats

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 10.2 ImageSurface

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 10.2.1 Formats

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 10.2.2 Loading images

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 10.2.3 Accessing the image data

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 10.3 SVGSurface

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 10.3.1 Other SVGSurface methods

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 10.4 PDFSurface

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 10.4.1 Multipage PDFs

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 10.4.2 Other PDFSurface methods

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 10.5 PSSurface

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 10.6 RecordingSurface

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 10.7 ScriptSurface

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.



## 10.8 TeeSurface

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 10.9 GUI surfaces

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

# 11 Integration with other libraries

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 11.1 How Pycairo stores image data

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 11.1.1 Data format

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 11.1.2 RGB24 data

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 11.1.3 Accessing data as bytes

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 11.1.4 ARGB32 data

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 11.1.5 Other formats

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 11.2 PIL (Pillow) integration

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 11.2.1 Reading a JPEG file

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 11.2.2 Writing a JPEG file

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 11.3 Numpy integration

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 11.3.1 Creating a numpy array from a surface

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 11.3.2 Creating a surface from a numpy array

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

# 12 Reference

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

## 12.1 Radians

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 12.1.1 What are radians

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 12.1.2 Conversion

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.

### 12.1.3 Key angles

This content is not available in the sample book. The book can be purchased on Leanpub at <http://leanpub.com/computergraphicsinpython>.