PDF::Writer for Ruby

Native Ruby PDF Document Creation

The Ruby PDF Project
http://rubyforge.org/projects/ruby-pdf
version 1.1.8

1. Introduction to PDF::Writer for Ruby

PDF::Writer is designed to provide a pure Ruby way to dynamically create PDF documents. Obviously, this will not be as fast as one that uses a compiled extension, but it is surprisingly fast. This manual is, in some ways, a worst case scenario because of the number of examples that must be displayed.

PDF::Writer does not yet implement the full PDF specification (any version), but it implements a substantial portion of it, with more to come. It also implements a more complex document layer on top of the portion that is implemented.

This manual (manual.pdf) is generated by PDF::TechBook using the application runner (bin/techbook) and the text version of the manual (manual.pwd). It is a comprehesive introduction to the use of PDF::Writer and a simple markup language interpreter. PDF::Writer itself only implements a few markup items, mostly relating to bold and italic text.

PDF::Writer is based on Adobe's PDF Reference, Fifth Edition, version 1.6. This and earlier editions are available from the Adobe website. The original implementation was a port of the public domain R&OS PDF Class for PHP.

Other demo programs are available in the **demo/** directory. Alternatively, they may be downloaded separately from the Ruby PDF Tools project on RubyForge.

Installation

If this manual was generated from a local installation of PDF::Writer, then congratulations! PDF::Writer is installed. Otherwise, you are reading a manual generated otherwise. If you want to install PDF::Writer, you can download it from the Ruby PDF Tools project on RubyForge or install it with RubyGems.

PDF::Writer has dependencies on Transaction::Simple 1.3.0 or later and color-tools 1.0.0 or later. These must be installed for PDF::Writer to work. RubyGems installation will automatically detect and offer to download and install these dependencies.

PDF::Writer is installed with:

```
% ruby setup.rb
```

Table of Contents

2. Making Documents with PDF::Writer

Document creation with PDF::Writer is quite easy. The following code will create a single-page document that contains the phrase “Hello, Ruby.” centered at the top of the page in 72-point (1”) type.

```
  # This code is demo/hello.rb
require "pdf/writer"
pdf = PDF::Writer.new
pdf.select_font "Times-Roman"
pdf.text "Hello, Ruby.", :font_size => 72, :justification => :center
File.open("hello.pdf", "wb") { |f| f.write pdf.render }
```

This one, on the other hand, uses a very (in)famous phrase and image. Note that the images are JPEG and PNG—PDF::Writer cannot use GIF images.

```
  # This code is demo/chunkybacon.rb
require "pdf/writer"
pdf = PDF::Writer.new
pdf.select_font "Times-Roman"
pdf.text "Chunky Bacon!!", :font_size => 72, :justification =>
:center
  # PDF::Writer#image returns the image object that was added.
i0 = pdf.image "../images/chunkybacon.jpg", :resize => 0.75
pdf.image "../images/chunkybacon.png",
          :justification => :center, :resize => 0.75
  # It can reinsert an image if wanted.
pdf.image i0, :justification => :right, :resize => 0.75
pdf.text "Chunky Bacon!!", :font_size => 72, :justification =>
:center
File.open("chunkybacon.pdf", "wb") { |f| f.write pdf.render }
```

If those aren’t enough to whet your appetite for how easy PDF::Writer can be, well, no chunky bacon for you!

3. Fundamental PDF Concepts

This section covers fundamental concepts to the creation of PDF documents that will assist in the understanding of the details that follow in later sections.

PDF Coordinate Space and User Units

PDF documents don't use inches or millimetres for measurement, and the coordinate space is slightly different than that used by other graphics canvases.

PDF User Units

PDF user units are by default points. A modern point is exactly 1/72" (1/72 inch); it is generally accepted that 72 point type is one inch in height. Historically, one point was 0.0138", or just under 1/72".

PDF User Unit Conversions

From	To	From	To
1 point	0.3528 mm	1 point	1/72"
10 mm	28.35 pts		
A4	210 mm × 297 mm	A4	595.28 pts × 841.89 pts
LETTER	8½" × 11"	LETTER	612 pts × 792 pts

It is possible to scale the user coordinates with PDF::Writer#scale_axis. With scale_axis(0.5, 0.5), each coordinate is ½ point in size (1/144") in both the x and y directions. See the following discussion on PDF Coordinate Space for the limitation of performing transformations on the coordinate axis.

PDF::Writer provides utility methods to convert between normal measurement units and and points. These utility methods are available either as class mehods or as instance methods.

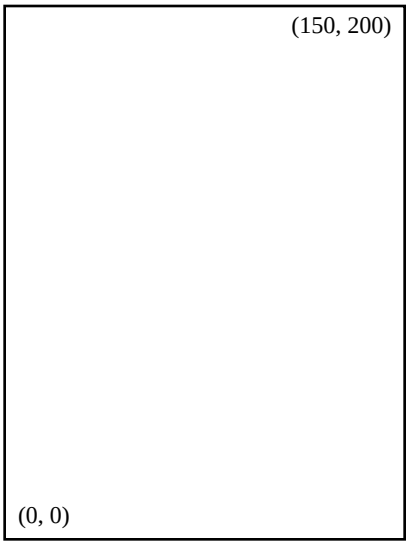
PDF::Writer.cm2pts(x)
Converts from centimetres to points.

PDF::Writer.in2pts(x)
Converts from inches to points.

PDF::Writer.mm2pts(x)
Converts from millimetres to points.

PDF Coordinate Space

The standard PDF coordinate space is a little different than might be expected. Most graphics canvases place the base coordinate (0, 0) in the upper left-hand corner of the canvas; the common PDF coordinate sytem places the base coordinate in the lower left-hand corner of the canvas.



PDF::Writer uses the standard coordinate space. Any transformations on the coordinate space (or,

more accurately, on the coordinate axis) may affect the usability of PDF::Writer. Standard transformations available against the coordinate space axis are rotations (counter-clockwise), scaling, translation (repositioning of the base coordinate) and skew. Each of these will be discussed in more detail in a later section of this manual, after other concepts have been introduced.

NOTE: As of PDF::Writer 1.1.0, angle rotations are now counter-clockwise, not clockwise as in earlier versions. This is a necessary incompatible change to make transformations more compatible with other vector graphics systems.

Fonts, Special Characters, and Character Maps in PDF::Writer

All PDF readers support fourteen standard fonts; twelve of these fonts are bold, italic (or oblique), and bold-italic (bold-oblique) variants of three base font families and the other two are symbolic fonts.

Default Fonts in PDF

Family	Name	Filename
Courier	Courier	Courier.afm
	Courier-Bold	Courier-Bold.afm
	Courier-Oblique	Courier-Oblique.afm
	Courier-BoldOblique	Courier-BoldOblique.afm
Helvetica	Helvetica	Helvetica.afm
	Helvetica-Bold	Helvetica-Bold.afm
	Helvetica-Oblique	Helvetica-Oblique.afm
	Helvetica-BoldOblique	Helvetica-BoldOblique.afm
Symbol	Symbol	Symbol.afm
Times-Roman	Times-Roman	Times-Roman.afm
	Times-Bold	Times-Bold.afm
	Times-Italic	Times-Italic.afm
	Times-BoldItalic	Times-BoldItalic.afm
ZapfDingbats	ZapfDingbats	ZapfDingbats.afm

In addition to these fourteen standard fonts, PDF supports the embedding of PostScript Type 1, TrueType, and OpenType fonts. PDF::Writer explicitly supports Type 1 and TrueType fonts. Inasmuch as OpenType fonts may be compatible with TrueType fonts, they may be supported. Embedded fonts require font metrics information. See “ **Embedding Fonts**”, “ *Type 1 Fonts*” and “ *TrueType Fonts*” below for more information.

PDF::Writer will find (or attempt to find) system font location(s) so that any existing Type 1 or TrueType fonts can be loaded directly from the system locations. These paths are found in **PDF::Writer::FONT_PATH**. By modifying this value, it is possible to tell PDF::Writer to find font files in alternate locations. Adobe font metrics (AFM) files are found in the paths described by **PDF::Writer::FontMetrics::METRICS_PATH**, the current directory, and the Ruby **$LOAD_PATH**.

Fonts are selected with PDF::Writer#select_font. This will load the appropriate font metrics file and—if the font is not one of the fourteen built-in fonts—search for the associated Type 1 or TrueType font to embed in the generated PDF document.

PDF::Writer#select_font(font_name, encoding = nil)

This selects, and possibly loads, a font to be used from this point in the document. The font name is as noted above. The format and the use of the **encoding** parameter is discussed extensively in " *Encoding Character Maps*".

The encoding directive will be effective **only when the font is first loaded**.

```
  # use a Times-Roman font with MacExpertEncoding
pdf.select_font("Times-Roman", "MacExpertEncoding")
  # this next line should be equivalent
pdf.select_font("Times-Roman", { :encoding => "MacExpertEncoding" })
  # Set up the Helvetica font for use with German characters as an offset
  # of the WinAnsiEncoding.
diff= {
  196 => "Adieresis",
  228 => "adieresis",
  214 => "Odieresis",
  246 => "odieresis",
  220 => "Udieresis",
  252 => "udieresis",
  223 => "germandbls"
}
pdf.select_font("Helvetica", { :encoding => "WinAnsiEncoding",
:differences => diff })
```

Font Families

It is possible to define font families in PDF::Writer so that text state translations may be performed. The only text state translations that PDF::Writer currently recognises are 'b' (bold), 'i' (italic), 'bi' (bold italic), and 'ib' (italic bold).

PDF::Writer#font_families

PDF::Writer#font_families is a Hash that maps the default font name (such as "Helvetica") to a mapping of text states and font names, as illustrated below.

```
pdf = PDF::Writer.new
  # Note: "bi" (<b><i>) can be implemented as a
  # different font than "ib" (<i><b>).
pdf.font_families["Helvetica"] =
{
  "b"   => "Helvetica-Bold",
  "i"   => "Helvetica-Oblique",
  "bi"  => "Helvetica-BoldOblique",
  "ib"  => "Helvetica-BoldOblique"
}
```

The use of font families will allow the in-line switching between text states and make it unnecessary to use **#select_font** to change between these fonts. This will also ensure that the encoding and differences (see " **Special Characters, Character Maps, and Unicode**" below) for the selected font family will be consistent. The default font families detailed above (for Helvetica, Times Roman, and Courier) have already been defined for PDF::Writer.

Embedding Fonts

PDF::Writer will properly embed both TrueType and PostScript Type 1 fonts, but these fonts will require Adobe Font Metrics (AFM) files (extension .afm). These files should exist for PostScript

Type 1 files but may not exist for TrueType files on any given system.

Type 1 Fonts

PostScript Type 1 fonts are fully supported by PDF, but only the binary (.pfb) forms of Type 1 fonts may be embedded into a PDF document. If only the ASCII (.pfa) form of the Type 1 font is available, then the program "t1binary" available as part of the Type 1 utilities package found at the LCDF type software page will create the binary form from the ASCII form. There is a suggestion on that page that older Macintosh Type 1 fonts will need additional conversion with one of the provided utilities.

TrueType Fonts

AFM files can be generated for TrueType fonts with the program "ttf2afm", which is part of the package pdfTex by Han The Thanh. In a future release of PDF::Writer or another program from the Ruby PDF project, this requirement should be eliminated.

Embedded Font Licensing Restrictions

As noted above, PDF::Writer will embed Type 1 or TrueType font programs in the PDF document. Fonts are recognised as copyrightable intellectual property in some jurisdictions. TrueType fonts encode some licensing rights in the font programs and PDF::Writer will check and warn if fonts not licensed for inclusion in a document are selected. It is up to the users of PDF::Writer to ensure that there are no licence violations of fonts embedded in the generated PDF documents.

Special Characters, Character Maps, and Unicode

Fonts in PDF documents can include encoding information. The PDF standard encodings are 'none', 'WinAnsiEncoding', 'MacRomanEncoding', or 'MacExpertEncoding'. Symbolic fonts should be encoded with the 'none' encoding. The default encoding used in PDF::Writer is 'WinAnsiEncoding'.

'WinAnsiEncoding' encoding is not quite the same as Windows code page 1252 (roughly equivalent to latin 1). Appendix D of the Adobe PDF Reference version 1.6 contains full information on all encoding mappings (including mappings for the two included symbolic fonts).

Encoding Character Maps

The main way of providing particular character support with PDF::Writer is through a differences map, or a character subtitution table. This is done only when initially selecting the font; it will not be reapplied after the font has been loaded from disk once (this limitation applies to the fourteen built-in fonts as well).

```
encoding = {
  :encoding     => "WinAnsiEncoding",
  :differences  => {
    215 => "multiply",
    148 => "copyright",
  }
}
pdf.select_font("Helvetica", encoding)
```

The above code maps the bytes 215 (0xd7) and 148 (0x94) to the named characters "multiply" and "copyright" in the Helvetica font. These byte values are the characters "©" and "×" in Windows 1252 but are undefined in the font and therefore require additional information to present and space these characters properly.

As of PDF::Writer version 1.1, these difference maps will be inserted into the PDF documents as well. This change is experimental but should be safe.

Unicode

PDF supports UTF-16BE encoding of strings—but each such string must begin with the UTF-16BE byte order mark (BOM) of U+FEFF (0xFE followed by 0xFF). PDF::Writer does not, at this point, explicitly support either UTF-8 or UTF-16BE strings. If all of the conditions are correct, the following code should display the Japanese characters for “Nihongo” (the name of the Japanese language). As of 2005.05.02, this will not work (at least in this manual).

```
pdf.text("\xfe\xff\x65\xe5\x67\x2c\x8a\x9e")
```

There are additional limitations and features for Unicode support, but a later version of PDF::Writer will support Unicode (at least UTF-8) documents.

4. Working with PDF Documents

The PDF::Writer class is used to create PDF documents in Ruby. It is a “smart” writing canvas on which both text and graphics may be drawn. 5. Graphics in PDF Documents covers the methods that are used to draw graphics in PDF::Writer. This chapter covers text drawing and most other PDF document operations.

The canvas provided by PDF::Writer is described as a “smart” canvas because it is aware of common text writing conventions including page margins and multi-column output.

Creating PDF Documents

There are two ways to create PDF documents with PDF::Writer. Both will provide instances of PDF::Writer; these are **PDF::Writer.new** and **PDF::Writer.prepress**.

PDF::Writer.new

This method creates a new PDF document as a writing canvas. Without any parameters, it will create a PDF version 1.3 document that uses pages that are standard US Letter (8½” × 11”) in portrait orientation. It accepts three named parameters, **:paper**, **:version**, and **:orientation**.

```
require "pdf/writer"
  # US Letter, portrait, 1.3
PDF::Writer.new
  # 1. A4, portrait, 1.3
PDF::Writer.new(:paper => "A4")
  # 2. 150cm × 200cm, portrait, 1.3
PDF::Writer.new(:paper => [ 150, 200 ])
  # 3. 150pt × 200pt, portrait, 1.3
PDF::Writer.new(:paper => [ 0, 0, 150, 200 ])
  # US Letter, landscape, 1.3
PDF::Writer.new(:orientation => :landscape)
  # US Letter, portrait, 1.5
PDF::Writer.new(:version => PDF_VERSION_15)
```

The **:paper** parameter specifies the size of a page in PDF::Writer. It may be specified as: (1) a standard paper size name (see the table “PDF::Writer Page Sizes” below for the defined paper size names); (2) a **[width, height]** array measured in centimetres; or (3) a **[x0, y0, x1, y1]** array measured in points where **(x0, y0)** represents the lower left-hand corner and **(x1, y1)** represent the upper right-hand corner of the page.

The **:orientation** parameter specifies whether the pages will be **:portrait** (the long edge is the height of the page) or **:landscape** (the long edge is the width of the page). These are the only allowable values.

The **:version** parameter specifies the minimum specification version that the document will adhere to. As of this version, the PDF document version is inserted but not used to control what features may be inserted into a document. The latest version of the PDF specification is PDF 1.6 (associated with Adobe Acrobat 7); it is recommended that the default version (1.3) be kept in most cases as that will ensure that most users will have access to the features in the document. A later version of PDF::Writer will include version controls so that if when creating a PDF 1.3 document, features from PDF 1.6 will not be available. PDF::Writer currently supports features from PDF 1.3 and does not yet provide access to the advanced features of PDF 1.4 or higher.

PDF::Writer Page Sizes

Type	Size	Type	Size
2A0	46.811" x 66.220" (118.9cm x 168.2cm)	4A0	66.220" x 93.622" (168.2cm x 237.8cm)
A0	33.110" x 46.811" (84.1cm x 118.9cm)	A1	23.386" x 33.110" (59.4cm x 84.1cm)
A2	16.535" x 23.386" (42cm x 59.4cm)	A3	11.693" x 16.535" (29.7cm x 42cm)
A4	8.268" x 11.693" (21cm x 29.7cm)	A5	5.827" x 8.268" (14.8cm x 21cm)
A6	4.134" x 5.827" (10.5cm x 14.8cm)	A7	2.913" x 4.134" (7.4cm x 10.5cm)
A8	2.047" x 2.913" (5.2cm x 7.4cm)	A9	1.457" x 2.047" (3.7cm x 5.2cm)
A10	1.024" x 1.457" (2.6cm x 3.7cm)		
B0	39.370" x 55.669" (100cm x 141.4cm)	B1	27.835" x 39.370" (70.7cm x 100cm)
B2	19.685" x 27.835" (50cm x 70.7cm)	B3	13.898" x 19.685" (35.3cm x 50cm)
B4	9.842" x 13.898" (25cm x 35.3cm)	B5	6.929" x 9.842" (17.6cm x 25cm)
B6	4.921" x 6.929" (12.5cm x 17.6cm)	B7	3.465" x 4.921" (8.8cm x 12.5cm)
B8	2.441" x 3.465" (6.2cm x 8.8cm)	B9	1.732" x 2.441" (4.4cm x 6.2cm)
B10	1.220" x 1.732" (3.1cm x 4.4cm)		
C0	36.102" x 51.063" (91.7cm x 129.7cm)	C1	25.512" x 36.102" (64.8cm x 91.7cm)
C2	18.032" x 25.512" (45.8cm x 64.8cm)	C3	12.756" x 18.032" (32.4cm x 45.8cm)
C4	9.016" x 12.756" (22.9cm x 32.4cm)	C5	6.378" x 9.016" (16.2cm x 22.9cm)
C6	4.488" x 6.378" (11.4cm x 16.2cm)	C7	3.189" x 4.488" (8.1cm x 11.4cm)
C8	2.244" x 3.189" (5.7cm x 8.1cm)	C9	1.575" x 2.244" (4.0cm x 5.7cm)
C10	1.102" x 1.575" (2.8cm x 4.0cm)		
EXECUTIVE	7.248" x 10.5" (18.410cm x 26.670cm)	FOLIO	8.5" x 13" (21.590cm x 33.020cm)
LEGAL	8.5" x 14" (21.590cm x 35.560cm)	LETTER	8.5" x 11" (21.590cm x 27.940cm)
RA0	33.858" x 48.032" (86cm x 122cm)	RA1	24.016" x 33.858" (61cm x 86cm)
RA2	16.929" x 24.016" (43cm x 61cm)	RA3	12.008" x 16.929" (30.5cm x 43cm)
RA4	8.465" x 12.008" (21.5cm x 30.5cm)		
SRA0	35.433" x 50.394" (90cm x 128cm)	SRA1	25.197" x 35.433" (64cm x 90cm)
SRA2	17.717" x 25.197" (45cm x 64cm)	SRA3	12.598" x 17.717" (32cm x 45cm)
SRA4	8.858" x 12.598" (22.5cm x 32cm)		

PDF::Writer.prepress

This is an alternative way to create a new PDF document. In addition to the named parameters in PDF::Writer.new (**:paper**, **:orientation**, and **:version**), this method accepts the following options as well.

:left_margin, **:right_margin**, **:top_margin**, and **:bottom_margin** specify the margins. Prepress marks are placed relative to the margins, so when creating a document in prepress mode the margins must be specified as the document is created. Future versions of PDF::Writer will be more flexible on this. The default margins are 36pts (about ½").

:bleed_size and **:mark_length** specify the size of the bleed area (default is 12 points) and the length of the prepress marks (default is 18 points).

Prepress marks will appear on all pages.

```
require "pdf/writer"
PDF::Writer.prepress # US Letter, portrait, 1.3, prepress
```

Adding Text to PDF Document

There are two different and complementary ways of adding text to a PDF document with PDF::Writer. The easier is to use the generated PDF as a "smart" canvas, where information internal to the document is used to maintain a current writing pointer (see " Internal Writing Pointer") and text is written within the writing space defined by document margins (see " Document Margins and Page Dimensions"). The harder is to place the text on the document canvas explicitly, ensuring that the text does not overflow the page horizontally or vertically.

In order to support more flexible and humane layout, PDF::Writer supports XML-like tags that can be used to change the current text state (discussed earlier in " **Font Families**"), substitute alternate text, or apply custom formatting. This is discussed in detail in " **Text Tags**".

Text Wrapping

PDF::Writer uses a very simple text wrapping formula. If a line looks like it's going to exceed its permitted width, then PDF::Writer backtracks to the previous hyphen ('-') or space to see if the text will then fit into the permitted width. It will do this until the text will fit. If no acceptable hyphen or space can be found, the text will be forcibly split at the largest position that will fit in the permitted width.

"Smart" Text Writing

As noted, PDF::Writer has two different ways to write text to a PDF document. This is expressed with three different methods. The method that uses the PDF document as a "smart" canvas is **PDF::Writer#text**.

PDF::Writer#text(text, options = nil)

This method will add text to the document starting at the current drawing position (described in " Internal Writing Pointer"). It uses the document margins (see " Document Margins and Page Dimensions") to flow text onto the page. If an explicit newline ('\n') is encountered, it will be used.

The text is drawn from the left margin until it reaches the right margin. If the text exceeds the right margin, then the text will be wrapped and continued on the next line. If the text to be written will exceed the bottom margin, then a new page will be created (detailed in " Pages and Multi-Column Output").

```
pdf.text("Ruby is fun!")
```

Text Size

If the **:font_size** is not specified, then the last font size or the default font size of 12 points will be used. If this value is provided, this **changes** the current font size just as if the #font_size attribute were set.

```
pdf.text("Ruby is fun!", :font_size => 16)
```

Text Justification

Text may be justified within the margins (normal or overridden) in one of four ways with the option

:justification.

- **:left** justification is also called “ragged-right”, because the text is flush against the left margin and the right edge of the text is uneven. This is the default justification.
- **:right** justification places the text flush agains the right margin; the left edge of the text is uneven.
- **:center** justification centers the text between the margins.
- **:full** justification ensures that the text is flush against both the left and right margins. Additional space is inserted between words to make this happen. The last line of a paragraph is only made flush left.

```
pdf.text("Ruby is fun!", :justification => :left)
pdf.text("Ruby is fun!", :justification => :right)
pdf.text("Ruby is fun!", :justification => :center)
pdf.text("Ruby is fun!", :justification => :full)
```

Margin Override Options

The drawing positions and margins may be modified with the **:left**, **:right**, **:absolute_left**, and **:absolute_right** options. The **:left** and **:right** values are PDF userspace unit offsets from the left and right margins, respectively; the **:absolute_left** and **:absolute_right** values are userspace units effectively providing *new* margins for the text to be written. Note that **:absolute_right** is offset from the left side of the page, not the right side of the page.

```
  # The left margin is shifted inward 50 points.
pdf.text("Ruby is fun!", :left => 50)
  # The right margin is shifted inward 50 points.
pdf.text("Ruby is fun!", :right => 50)
  # The text is drawn starting at 50 points from the left of the page.
pdf.text("Ruby is fun!", :absolute_left => 50)
  # The text is drawn ending at 300 points from the left of the page.
pdf.text("Ruby is fun!", :absolute_right => 300)
```

Line Spacing Options

Normally, each line of text is separated only by the descender space for the font. This means that each line will be separated by only enough space to keep it readable. This may be changed by changing the line spacing (**:spacing**) as a multiple of the normal line height (authors’ manuscripts are often printed double spaced or **:spacing => 2**). This can also be changed by redefining the total height of the line (the *leading*) independent of the font size. With **:font_size => 12**, both **:spacing => 2** and **:leading => 24** do the same thing. The **:leading** value overrides the **:spacing** value.

```
  # These are the same
pdf.text("Ruby is fun!", :spacing => 2)
pdf.text("Ruby is fun!", :leading => 24)
```

Test Writing

Not generally used, the **:test** option can be set to **true**. This will prevent the text from being written to the page and the method will return **true** if the text will be overflowed to a new page or column.

Explicit Text Placement

Text can also be placed starting with specific X and Y coordinates using either **PDF::Writer#add_text_wrap** or **PDF::Writer#add_text**.

PDF::Writer#add_text_wrap(x, y, width, text, size = nil, justification = :left, angle = 0, test = false)

This will add text to the page at position **(x, y)**, but ensures that it fits within the provided **width**. If it does not fit, then as much as possible will be written using the wrapping rules on page 11. The remaining text to be written will be returned to the caller.

```
rest = pdf.add_text_wrap(150, pdf.y, 150, "Ruby is fun!", 72)
```

If **size** is not specified, the current #font_size will be used. This parameter has changed position with **text** in PDF::Writer 1.1. If your code appears to be using the old parameter call, it will be warned.

The optional justification parameter works just as described on page 11, except that instead of margin width, it is the boundaries defined by the **x** and **width** parameters.

Text may be drawn at an angle. This will be discussed fully in the section on **PDF::Writer#add_text**.

If the **test** parameter is **true**, the text will not be written, but the remainder of the text will be returned.

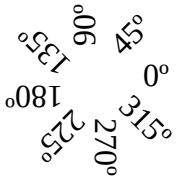
PDF::Writer#add_text(x, y, text, size = nil, angle = 0, word_space_adjust = 0)

This places the full text string at the **(x, y)** position and the specified **angle** (measured in degrees of a circle, in a counter-clockwise direction) with respect to the angle of the coordinate axis.

NOTE: As of PDF::Writer 1.1.0, angle rotations are now counter-clockwise, not clockwise as in earlier versions. This is a necessary incompatible change to make transformations more compatible with other vector graphics systems.

If **size** is not specified, the current #font_size will be used. This parameter has changed position with **text** in PDF::Writer 1.1. If your code appears to be using the old parameter call, it will be warned.

```
0.step(315, 45) do |angle|
  pdf.add_text(pdf.margin_x_middle, pdf.y, "#{angle}º".rjust(8), 12,
               angle)
end
```



The **adjust** parameter adjusts the space between words by the specified number of PDF userspace units. This is primarily used by PDF::Writer#text and PDF::Writer#add_text_wrap to support text justification.

Text Tags

Text is written in PDF documents as simple strings positioned on the page canvas. Unlike HTML, there is no easy way to indicate that the font or the font size should be changed. This has to be

managed entirely by the generating application. PDF::Writer has a mechanism called “text tags” to manage these changes and provide hooks to introduce new behaviours.

PDF::Writer provides five classes of text tags. Two are used to change font families and have fixed meaning; the other three are callback tags.

Because text tags use XML semantics, XML escape characters are necessary to display tag characters.

- < => <
- > => >
- & => &

Font Family Tags (<b> and <i>)

The font family tags have fixed meaning and will add the style identifiers ‘b’ and ‘i’ to the current text state, respectively. Both require closing tags. The text between the tags will be rendered in **bold**, *italic*, or ***bold italic*** versions of the master font as appropriate. If the font does not exist, then the master font is used: PDF::Writer will not simulate italics or boldface.

See **Font Families** for an explanation of font families and text state.

Callback Tags and Parameters

The remaining three tag classes are callback tags. All callback tags have a regular form of **<*x*:*name*[*parameters*]>**. This is essentially the form of a namespaced XML tag. Parameters must be specified as with XML, in the form of ***name*="*value*"** or ***name*='*value*'**. Parameters will be provided to callbacks as a name-value hash:

```
<c:alink uri="http://rubyforge.org/">RubyForge</c:alink>
```

When the opening *c:alink* tag is encountered, the callback will receive a tag parameter hash of:

```
{ "uri" => "http://rubyforge.org/" }
```

When the closing *c:alink* tag is encountered, the callback will receive an empty parameter hash (**{}**).

Tag and Callback Class Association

The constant hash PDF::Writer::TAGS has three hashes that keep track of tag name and callback class associations. A sample set of associations might be:

```
TAGS = {
  :pair     => { "alink"  => PDF::Writer::TagAlink, },
  :single   => { "disc"   => PDF::Writer::TagDisc, },
  :replace  => { "xref"   => PDF::TechBook::TagXref, },
}
```

Callback tags must define **#[](pdf, params)**. The standard callback classes define **CallbackClass.[](pdf, params)** so that the callbacks are used without instantiation.

Replacement Tags

Replacement tags will replace the tag with a value computed by the callback. It may perform additional processing, but no location information is provided to the callback. The return from this callback **must** be the string that will replace the tag. A replacement tag looks like this in text:

```
<r:xref name="FontFamilies" label="title" />
```

Replacement tags always begin with the “namespace” of ‘r:’ and do not surround text (they are stand-alone tags).

The **params** parameter sent to **#[]** is the default parameters hash, containing only information from

the tag itself.

Sample Replacement Tag

The example below is the “xref” tag from PDF::TechBook.

```
class TagXref
  def self.[](pdf, params)
    name  = params["name"]
    item  = params["label"]
    xref = pdf.xref_table[name]
    if xref
      case item
      when 'page'
        label = xref[:page] || xref[:label]
      when 'label'
        label = xref[:label]
      end
      "<c:ilink dest='#{xref[:xref]}'>#{label}</c:ilink>"
    else
      warn PDF::Writer::Lang[:techbook_unknown_xref] % [ name ]
      PDF::Writer::Lang[:techbook_unknown_xref] % [ name ]
    end
  end
end
PDF::Writer::TAGS[:replace]["xref"] = PDF::TechBook::TagXref
```

Single Drawing Tags

Single drawing tags are stand-alone tags that will perform drawing behaviours when the tag is encountered. Location information is provided to the callback and the callback may return a hash with **(x, y)** information to adjust the position of following text. A single drawing tag looks like this in text:

```
<C:bullet />
```

Single drawing tags always begin with the “namespace” of ‘C:’ and do not surround text. The **params** parameter sent to **#[]** is a complex compound object. See *Drawing Tag Parameters* for more information on the parameters provided to drawing tags.

A single drawing tag callback will be called once and only once when it is encountered.

Sample Single Drawing Tag

The example below is the “bullet” tag from PDF::Writer.

```
class TagBullet
  DEFAULT_COLOR = Color::RGB::Black
  class << self
    attr_accessor :color
    def [](pdf, info)
      @color ||= DEFAULT_COLOR
      desc  = info[:descender].abs
      xpos  = info[:x] - (desc * 2.00)
      ypos  = info[:y] + (desc * 1.05)
      pdf.save_state
      ss = StrokeStyle.new(desc)
      ss.cap  = :butt
      ss.join = :miter
      pdf.stroke_style ss
      pdf.stroke_color @style
      pdf.circle_at(xpos, ypos, 1).stroke
      pdf.restore_state
    end
  end
```

```
end
TAGS[:single]["bullet"] = TagBullet
```

Paired Drawing Tags

Paired drawing tags are tags that surround text and perform drawing behaviours realted to that text when the tag is encountered. Location information is provided to the callback and the callback may return a hash with **(x, y)** information to adjust the position of following text. A paire ddrawing tag looks like this in text:

```
<c:uline>text</c:uline>
```

Paired drawing tags always begin with the "namespace" of 'c:' and surround text. The **params** parameter sent to **#[]** is a complex compound object. See *Drawing Tag Parameters* for more information on the parameters provided to drawing tags.

A paired drawing tag callback will be called at least twice, and may be called many times while it is open. It will be called once when the tag is opened, once for every line "end" that is reached, once for every line "start" that is reached, and once when the tag is closed. For PDF::Writer#text, line start and end positions are whenever a natural or wrapped newline is encountered. For PDF::Writer#add_text_wrap and PDF::Writer#add_text, the line start and end are used for **each call** while a tag is open.

Sample Paired Drawing Tag

The example below is the "uline" tag from PDF::Writer.

```
class TagUline
  DEFAULT_STYLE = {
    :factor  => 0.05
  }
  class << self
    attr_accessor :style
    def [](pdf, info)
      @style ||= DEFAULT_STYLE.dup
      case info[:status]
      when :start, :start_line
        @links ||= {}
        @links[info[:cbid]] = {
          :x         => info[:x],
          :y         => info[:y],
          :angle     => info[:angle],
          :descender => info[:descender],
          :height    => info[:height],
          :uri       => nil
        }
        pdf.save_state
        pdf.stroke_style! StrokeStyle.new(info[:height] * @style[:factor])
      when :end, :end_line
        start = @links[info[:cbid]]
        theta = PDF::Math.deg2rad(start[:angle] - 90.0)
        drop  = start[:height] * @style[:factor] * 1.5
        drop_x = Math.cos(theta) * drop
        drop_y = -Math.sin(theta) * drop
        pdf.move_to(start[:x] - drop_x, start[:y] - drop_y)
        pdf.line_to(info[:x] - drop_x, info[:y] - drop_y).stroke
        pdf.restore_state
      end
    end
  end
end
TAGS[:pair]["uline"] = TagUline
```

Drawing Tag Parameters

Drawing tags expect more information than is provided to replacement tags, as they are expected to draw something or perform other tasks on the document at the time that they are encountered in the appropriate position(s). The parameters provided are described below.

:x
The current **x** position of the text.

:y
The current **y** position of the text.

:angle
The current **angle** of the text in *degrees*. This will allow the correct calculation of text or drawing orientation.

:params
The hash of named parameters for the tag. This is the same as the value that is provided to replacement tags.

:status
One of the values **:start**, **:end**, **:start_line**, or **:end_line**

- **:start** is provided to the callback when encountering the opening tag for a paired drawing tag, or when encountering a single drawing tag.
- **:end** is provided to the callback when the closing tag for a paired drawing tag.
- **:start_line** is provided to the callback when a new line is to be drawn and a paired tag is open.
- **:end_line** is provided to the callback when a line is finished writing and a paired tag is open.

:cbid
The callback identifier; this may be used as a key into a variable which keeps state for the various callbacks. In the “uline” example, the **@links** variable keeps this information.

:callback
The name of the tag, used to find the callback object. This is only set for opening paired tags or single tags.

:height
The font height at the time the tag was encountered.

:descender
The font descender at the time the tag was encountered.

Known Text Tags

A few useful text tags and their callbacks have been defined in PDF::Writer. Two others are used by the PDF::TechBook class used to generate this manual.

<c:alink uri="URI">
The <c:alink uri="URI"> tag is used to make a link to a resource external to the document. This can be any URL handler registered by the operating system for processing. The text “<c:alink uri="http://rubyforge.org/">RubyForge</c:alink>” will generate a hyperlink

like “ RubyForge”. This is known to work with HTTP, HTTPS, FTP, and MAILTO style URIs.

The callback object is **PDF::Writer::TagAlink**. The display style of linking and link underlining may be modified through **TagAlink.style**, which is a hash of five keys:

- **:color** is colour of the link underline. The default is Color::RGB::Blue. If nil, the current stroke colour will be maintained.
- **:text_color** is the colour of the text. The default is Color::RGB::Blue. If nil, the current fill colour will be maintained.
- **:factor** is the size of the line, as a multiplier of the text height. The default is 5% of the line height (0.05).
- **:line_style** is a style modification hash provided to PDF::Writer::StrokeStyle.new. The default is a solid line with normal cap, join, and miter limit values. See “ *PDF::Writer::StrokeStyle*” for more information.
- **:draw_line** indicates whether the underline should be drawn or not.

<c:ilink dest="DEST">

The <c:ilink dest="DEST"> tag is used to make a link to a destination within the document. Destinations must be created manually.

```
  # Code writing ...
  # ... text, pages, etc.
pdf.add_destination("x3y3z3", "Fit")
  # More code writing ...
  # ... text, pages, etc.
pdf.text("<c:ilink dest='x3y3z3'>Internal Link</c:ilink>")
```

This manual contains an extensive cross-reference table. This link will goes to the fifth item in that cross-reference table. (The code was <c:ilink dest="xref5">This link</c:ilink>.)

This is implemented with **PDF::Writer::TagIlink**. There are no configuration options.

<c:uline>

The <c:uline> tag is used to underline text between the opening tag and the closing tag. Therefore, “The quick brown fox <c:uline>is tired of</c:uline> jumping over the lazy dog” becomes “The quick brown fox is tired of jumping over the lazy dog”.

This is implemented with **PDF::Writer::TagUline**. The display style of linking and link underlining may be modified through **TagUline.style**, which is a hash of three keys:

- **:color** is colour of the link underline. The default is nil, which means the current stroke colour will be used.
- **:factor** is the size of the line, as a multiplier of the text height. The default is 5% of the line height (0.05).
- **:line_style** is a style modification hash provided to PDF::Writer::StrokeStyle.new. The default is a solid line with normal cap, join, and miter limit values. See “ *PDF::Writer::StrokeStyle*” for more information.

<C:bullet>

The <C:bullet> tag inserts a solid circular bullet, like this: “• ”. Bullets are implemented in **PDF::Writer::TagBullet**. The bullet display colour may be modified through **TagBullet.color**, which defaults to **Color::RGB::Black**.

<C:disc>

The <C:disc> tag inserts a hollow circular bullet, like this: "⊙". Bullets are implemented in **PDF::Writer::TagDisc**. The disc display colours may be modified throguh **TagDisc.foreground** and **TagDisc.background**, which default to **Color::RGB::Black** and **Color::RGB::White**, respectively.

<C:tocdots ...>
This is a stand-alone callback that draws a dotted line over to the right and appends a page number; it is implemented and used in **PDF::TechBook::TagTocDots**. It is of limited configurability in this version.

<r:xref name="XREFNAME" label="page|label" text="TEXT"/>
This replacement callback returns an internal link (<c:ilink...>) with either the name (label="title") or the page number (label="page") of the cross-reference, or an indicator for arbitrary text (label="text") drawn from the text attribute (e.g., text="random text here"). The page number will only be used if it is known at the time that the <r:xref> tag is processed; forward cross-references will always use the text (if present) or the label. This is implemented through **PDF::TechBook::TagXref**.

Text Height and Width

These methods return the height of the font or the width of the text.

PDF::Writer#font_height(font_size = nil)

Returns the height of the current font for the given size, measured in PDF userspace units. This is the distance from the bottom of the descender to the top of the capitals or ascenders. Uses the current #font_size if size is not provided.

PDF::Writer#font_descender(font_size = nil)

Returns the size of the descender—the distance below the baseline of the font—which will normally be a negative number. Uses the current #font_size if size is not provided.

PDF::Writer#text_width(text, font_size = nil)
PDF::Writer#text_line_width(text, font_size = nil)

Returns the width of the given text string at the given font size using the current font, or the current default font size if none is given. The difference between #text_width and #text_line_width is that the former will give the width of the largest line in a multiline string.

Document Margins and Page Dimensions

A document canvas should not be written from edge to edge in most cases; printers that try to print these items will often lose portions. From a design perspective, margins increase the amount of whitespace on the page and make the page easier to read. Drawing methods that do not use the internal writing pointer (see below) will not use the margin. Typically, these are methods that explicitly specify *(x, y)* coordinates. The default margins are 36pts (about ½").

Setting Margins

PDF::Writer provides four methods to quickly define all margins at once. With all four of these methods, if only one value is provided, all four margins are that value. Two values define the top/bottom margins and the left/right margins. Three values set separate top/bottom margins, but the left/right margins will be the same. Four values defines each margin independently, as shown

below. The only difference between the methods is the measurements used.

```
                                # T  L  B  R
pdf.margins_pt(36)              # 36 36 36 36
pdf.margins_pt(36, 54)          # 36 54 36 54
pdf.margins_pt(36, 54, 72)      # 36 54 72 54
pdf.margins_pt(36, 54, 72, 90)  # 36 54 72 90
```

PDF::Writer#margins_pt(top, left = top, bottom = top, right = left)
Set margins in points.

PDF::Writer#margins_cm(top, left = top, bottom = top, right = left)
Set margins in centimetres.

PDF::Writer#margins_mm(top, left = top, bottom = top, right = left)
Set margins in millimetres.

PDF::Writer#margins_in(top, left = top, bottom = top, right = left)
Set margins in inches.

PDF::Writer#top_margin, PDF::Writer#top_margin=
PDF::Writer#left_margin, PDF::Writer#left_margin=
PDF::Writer#bottom_margin, PDF::Writer#bottom_margin=
PDF::Writer#right_margin, PDF::Writer#right_margin=
In addition, each margin may be examined and modified independently, but all measurements for direct margin access are in PDF userspace units only.

```
pdf.top_margin      # -> 36
pdf.top_margin = 72
  # Also #left_margin, #bottom_margin, #right_margin
```

Using Margins and Page Dimensions

Margin values in PDF::Writer are offset values. If the right margin is ½", the absolute position of the right margin will be the width of the page less the offset of the right margin (conventionally, it will be the same as "pdf.page_width - pdf.right_margin". PDF::Writer provides attributes to read these and other page dimension values. All measurements are in PDF userspace units.

PDF::Writer#page_width
The width of the page.

PDF::Writer#page_height
The height of the page.

PDF::Writer#absolute_left_margin
The absolute horizontal position of the left margin.

PDF::Writer#absolute_right_margin
The absolute horizontal position of the right margin.

PDF::Writer#absolute_top_margin

The absolute vertical position of the top margin.

PDF::Writer#absolute_bottom_margin
The absolute vertical position of the bottom margin.

PDF::Writer#margin_height
The height of the writing space.

PDF::Writer#margin_width
The width of the writing space.

PDF::Writer#absolute_x_middle
The horizontal middle of the page based on the page dimensions.

PDF::Writer#absolute_y_middle
The vertical middle of the page based on the page dimensions.

PDF::Writer#margin_x_middle
The horizontal middle of the page based on the margin dimensions.

PDF::Writer#margin_y_middle
The vertical middle of the page based on the margin dimensions.

```
x = pdf.page_width - pdf.right_margin # flush right
y = pdf.page_height - pdf.top_margin  # flush top
  # Draw a box at the margin positions.
x = pdf.absolute_left_margin
w = pdf.absolute_right_margin - x
  # or pdf.margin_width
y = pdf.absolute_bottom_margin
h = pdf.absolute_top_margin - y
  # or pdf.margin_height
pdf.rectangle(x, y, w, h).fill
```

Internal Writing Pointer

PDF::Writer keeps an internal writing pointer for use with several different (mostly text) drawing methods. There are several ways to manipulate this pointer directly. The writing pointer value is changed automatically with drawing methods, column support, and starting new pages with #start_new_page.

PDF::Writer#y, PDF::Writer#y=

The vertical position of the writing point. The vertical position is constrained between the top and bottom margins. Any attempt to set it outside of those margins will cause the writing point to be placed absolutely at the margins.

```
pdf.y             # => 40
pdf.bottom_margin # => 36
pdf.y = 30        # => 36
```

PDF::Writer#pointer, PDF::Writer#pointer=

The vertical position of the writing point. If the vertical position is outside of the bottom margin, a

new page will be created.

```
pdf.pageset.size  # => 1
pdf.pointer       # => 40
pdf.bottom_margin # => 36
pdf.top_margin    # => 36
pdf.page_height   # => 736
pdf.pointer = 30
pdf.pageset.size  # => 2
pdf.pointer       # => 700
```

PDF::Writer#move_pointer(dy, make_space = false)

Used to change the vertical position of the writing point. The pointer is moved **down** the page by *dy*. If the pointer is to be moved up, a negative number must be used.

Moving up the page will not move to the previous page because of limitations in the way that PDF::Writer works. The writing point will be limited to the top margin position.

```
pdf.move_pointer(10)
```

If make_space is true and a new page is required to move the pointer, then the pointer will be moved down on the new page. This will allow space to be reserved for graphics. The following will guarantee that there are 100 units of space above the final writing point.

```
move_pointer(100, true)
```

Pages and Multi-Column Output

While it can be said that the PDF document is a canvas, it is a canvas of multiple pages. PDF::Writer does some intelligent management of the document to create pages as text flows off of it, but sometimes it will be necessary or useful to start pages manually. The class also provides facilities to write multi-column output automatically.

If a margin-aware drawing method reaches the bottom margin, it will request a new page with #start_new_page.

First Page

When a PDF::Writer document is first created, an initial page is created and inserted in the document. This page is always accessible with the #first_page method.

PDF::Writer#first_page

The first page created during startup, useful for adding something to it later.

Starting New Pages

PDF::Writer#start_new_page(force = false)

Nominally starts a new page and moves the writing pointer is moved back to the top margin. If multi-column output is on, a new column may be started instead of a new page. If an actual new page is required, the new page is added to the page list according to the page insert options (detailed below). If force is true, then a new page will be created even if multi-column output is on.

PDF::Writer#new_page(insert = false, page = nil, pos = :after)

Add a new page to the document. This also makes the new page the current active object. This allows for mandatory page creation regardless of multi-column output. Note the writing pointer position is not reset with the use of this method. For most purposes, #start_new_page is preferred.

Page Insertion Options

These options control where in the page set new pages are inserted when using PDF::Writer#start_new_page. When #insert_mode is on, new pages will be inserted at the specified page. If a page is inserted :before the page, then the #insert_position is changed to :after upon successful insertion of that page. This will ensure that if with a page list like "1 2 3 4 5 6", inserting a page between pages 3 and 4 with:

```
pdf.insert_mode     :on
pdf.insert_page     4
pdf.insert_position :before
 ... # insert pages 7, 8, and 9
```

will result in a page list like "1 2 3 7 8 9 4 5 6" and not "1 2 3 9 8 7 4 5 6."

When #insert_mode is off, pages are appended to the end of the document.

PDF::Writer#insert_mode(options = {})

Changes page insert mode. May be called as follows:

```
pdf.insert_mode         # => current insert mode
  # The following four affect the insert mode without changing the insert
  # page or insert position.
pdf.insert_mode(:on)    # enables insert mode
pdf.insert_mode(true)   # enables insert mode
pdf.insert_mode(:off)   # disables insert mode
pdf.insert_mode(false)  # disables insert mode
  # Changes the insert mode, the insert page, and the insert position at
the
  # same time. This is the same as calling:
  #
  #   pdf.insert_mode(:on)
  #   pdf.insert_page(:last)
  #   pdf.insert_position(:before)
opts = {
  :on       => true,
  :page     => :last,
  :position => :before
}
pdf.insert_mode(opts)
```

PDF::Writer#insert_page(page = nil)

Returns or changes the insert page property. The page must be either :last or an integer value representing the position in the page set—this value is completely unrelated to any page numbering scheme that may be currently in progress.

```
pdf.insert_page         # => current insert page
pdf.insert_page(35)     # insert at page 35
pdf.insert_page(:last)  # insert at the last page
```

PDF::Writer#insert_position(position = nil)

Returns or changes the insert position to be before or after the insert page.

```
pdf.insert_position           # => current insert position
pdf.insert_position(:before)  # insert before #insert_page
pdf.insert_position(:after)   # insert before #insert_page
```

Multi-Column Output

PDF::Writer#start_columns(size = 2, gutter = 10)

Starts multi-column output. Creates size number of columns with a gutter of PDF unit space between each column.

If columns are already started, this will return false as only one level of column definitions may be active at any time.

```
pdf.start_columns
pdf.start_columns(3)
pdf.start_columns(3, 2)
pdf.start_columns(2, 20)
```

When columns are on, #start_new_page will start a new column unless the last column is the current writing space, where it will create a new page.

PDF::Writer#stop_columns

Turns off multi-column output. If we are in the first column, or the lowest point at which columns were written is higher than the bottom of the page, then the writing pointer will be placed at the lowest point. Otherwise, a new page will be started.

PDF::Writer#column_width

Returns the width of the currently active column or 0 if columns are off.

PDF::Writer#column_gutter

Returns the size of the gutter between columns or 0 if columns are off.

PDF::Writer#column_number

Returns the current column number or 0 if columns are off. Column numbers are 1-indexed.

PDF::Writer#column_count

Returns the total number of columns or 0 if columns are off.

PDF::Writer#columns?

Returns true if columns are turned on.

Page Numbering

PDF::Writer supports automatic page numbering. The current page numbering system does not support anything except standard Arabic numbering (e.g., 1, 2, 3). The page numbering mechanism will be changing in a future version of PDF::Writer to be far more flexible.

PDF::Writer#start_page_numbering(x, y, size, pos = nil, pattern = nil, starting = nil)

Prepares the placement of page numbers on pages from the current page. Place them relative to the coordinates (x, y) with pos as the relative position. pos may be :left, :right, or :center. The page numbers will be written on each page using pattern.

When pattern is rendered, <PAGENUM> will be replaced with the current page number; <TOTALPAGENUM> will be replaced with the total number of pages in the page numbering scheme. The default pattern is “<PAGENUM> of <TOTALPAGENUM>”.

If starting is non-nil, this is the first page number. The number of total pages will be adjusted to account for this.

Each time page numbers are started, a new page number scheme will be started. The scheme number will be returned for use in other page number methods.

The following code produces a ten page document, numbered from the second page (labelled ‘1 of 9’) until the eighth page (labelled ‘7 of 9’).

```
pdf = PDF::Writer.new
pdf.select_font "Helvetica"
                    # page  1, blank
pdf.start_new_page  # page  2, 1 of 9
pdf.start_page_numbering(300, 500, 20, nil, nil, 1)
pdf.start_new_page  # page  3, 2 of 9
pdf.start_new_page  # page  4, 3 of 9
pdf.start_new_page  # page  5, 4 of 9
pdf.start_new_page  # page  6, 5 of 9
pdf.start_new_page  # page  7, 6 of 9
pdf.start_new_page  # page  8, 7 of 9
pdf.stop_page_numbers
pdf.start_new_page  # page  9, blank
pdf.start_new_page  # page 10, blank
```

Multiple page numbering schemes can be used on the same page, as demonstrated in demo/pagenumber.rb or below.

```
pdf = PDF::Writer.new
                    # Page  1: blank
sa = pdf.start_page_numbering(5, 60, 9, nil, nil, 1)
pdf.start_new_page  # Page  2: 1 of 2
pdf.start_new_page  # Page  3: 2 of 2
pdf.stop_page_numbering(true, :current, sa)
pdf.start_new_page  # Page  4: blank
sb = pdf.start_page_numbering(5, 50, 9, :center, nil, 10)
pdf.start_new_page  # Page  5: 10 of 12
pdf.start_new_page  # Page  6: 11 of 12
pdf.stop_page_numbering(true, :next, sb)
pdf.start_new_page  # Page  7: 12 of 12
sc = pdf.start_page_numbering(5, 40, 9, nil, nil, 1)
pdf.start_new_page  # Page  8: 1 of 3
pdf.start_new_page  # Page  9: 2 of 3
pdf.start_new_page  # Page 10: 3 of 3
pdf.stop_page_numbering(true, :current, sc)
pdf.start_new_page  # Page 11: blank
sd = pdf.start_page_numbering(5, 30, 9, nil, nil, 1)
pdf.start_new_page  # Page 12: 1 of 6
pdf.start_new_page  # Page 13: 2 of 6
se = pdf.start_page_numbering(5, 20, 9, nil, nil, 5)
sf = pdf.start_page_numbering(5, 10, 9, :right, nil, 1)
pdf.start_new_page  # Page 14: 3 of 6, 5 of 10, 1 of 8
pdf.start_new_page  # Page 15: 4 of 6, 6 of 10, 2 of 8
pdf.start_new_page  # Page 16: 5 of 6, 7 of 10, 3 of 8
pdf.stop_page_numbering(true, :next, sd)
```

```
pdf.start_new_page  # Page 17: 6 of 6, 8 of 10, 4 of 8
pdf.start_new_page  # Page 18: 9 of 10, 5 of 8
pdf.stop_page_numbering(true, :next, se)
pdf.stop_page_numbering(false, :current, sf)
pdf.start_new_page  # Page 19: 10 of 10
pdf.start_new_page  # Page 20: blank
```

Page	Contents	Page	Contents	Page	Contents	Page	Contents
1	blank	6	11 of 12	11	blank	16	5 of 6, 7 of 10, 3 of 8
2	1 of 2	7	12 of 12	12	1 of 6	17	6 of 6, 8 of 10, 4 of 8
3	2 of 2	8	1 of 3	13	2 of 6	18	9 of 10, 5 of 8
4	blank	9	2 of 3	14	3 of 6, 5 of 10, 1 of 8	19	10 of 10
5	10 of 12	10	3 of 3	15	4 of 6, 6 of 10, 2 of 8	20	blank

PDF::Writer#stop_page_numbering(stop_total = false, stop_at = :current, scheme = 0)

Stops page numbering for the provided scheme. Returns false if page numbering is off. If stop_total is true, then the totaling of pages for this page numbering scheme will be stopped as well. If stop_at is :current, then the page numbering will stop at the current page; otherwise, it will stop at the next page.

PDF::Writer#which_page_number(page_num, scheme = 0)

Given a particular generic page number page_num (numbered sequentially from the beginning of the page set), return the page number under a particular page numbering scheme. Returns nil if page numbering is not turned on.

Repeating Elements

It is common in documents to see items repeated from page to page. These may be watermarks, headers, footers, or other elements that must appear on multiple pages—aside from page numbering. PDF::Writer supports this through a mechanism called “loose content objects.”

Loose Content Objects

Up until this point, a page has been presented as the only canavas available to write and draw upon. This is a useful fiction, but it is a fiction. Any contents object may be drawn upon, and an implicit contents object is created when a new page is created. The methods discussed below create a new canvas for writing. It has the same physical boundaries as the page itself, and should only be written to with explicit locations, as #text is not aware of non-page canvases.

PDF::Writer#open_object

Makes a loose content object. Output will go into this object until it is closed. This object will not appear until it is included within a page. The method will return the object reference. To aid in the conceptual grouping of modifications to a loose content object, this method will yield the opened object if a block is provided.

PDF::Writer#reopen_object(id)

Makes the object for current content insertion the object specified by id, which is a value returned by either #open_object or #new_page.

PDF::Writer#close_object

Closes the currently open loose content object, preventing further writing against the object.

Using Loose Content Objects

Once a loose contents object has been created, it must be added to the collection of loose objects before it can be seen in the PDF document. PDF::Writer makes it easy to write these objects to the contents of pages when the pages are created.

PDF::Writer#add_object(id, where = :this_page)

After a loose content object has been created, it will only show if it has been added to a page or page(s) with this method. Where the loose content object will be added is controlled by the where option.

The object will not be added to itself.

- :this_page will add the object just to the current page.
- :all_pages will add the object to the current and all following pages.
- :even_pages will add the object to following even pages, including the current page if it is an even page.
- :odd_pages will add the object to following odd pages, including the current page if it is an odd page.
- :all_following_pages will add the object to the next page created and all following pages.
- :following_even_pages will add to the next even page created and all following even pages.
- :following_odd_pages will add to the next odd page created and all following odd pages.

PDF::Writer#stop_object(id)

Stops adding the specified object to pages after this page.

Example

The following example is used to create the heading of this manual.

```
pdf.open_object do |heading|
  pdf.save_state
  pdf.stroke_color! Color::Black
  pdf.stroke_style! PDF::Writer::StrokeStyle::DEFAULT
  s = 6
  t = "PDF::Writer for Ruby ~ Manual"
  w = pdf.text_width(t, s) / 2.0
  x = pdf.margin_x_middle
  y = pdf.absolute_top_margin
  pdf.add_text(x - w, y, t, s)
  x = pdf.absolute_left_margin
  w = pdf.absolute_right_margin
  y -= (pdf.font_height(s) * 1.01)
  pdf.line(x, y, w, y).stroke
  pdf.restore_state
  pdf.close_object
  pdf.add_object(heading, :next_all_pages)
end
```

Active Document Elements

Hyperlinks—to locations within the document or on the Internet—can be created in any document either using the callback forms (<c:alink> and <c:ilink>) for words, or #add_link and #add_internal_link for any other location in a document. Before internal links will work, the destination to which each refers must be created with #add_destination. In fact, a PDF document will not render if there is an internal link reference to a nonexistant destination. For those familiar

with HTML, this is similar to the following:

```
<h1 id="title">Title of Document</h1>
...
<a href="#title">Top</a>
```

PDF::Writer#add_destination(label, style, *params)

This method is used to create a labelled destination at this point in the document. The label is a string containing the name which will be used for links internal to the document (created by <c:ilink> or

The destination style controls how many parameters are required.

XYZ, 3: left, top, zoom

The viewport will be opened at position (left, top) with zoom percentage. If the values are the string "null", the current parameter values are unchanged.

Fit, 0

Fit the page to the viewport (horizontal and vertical). There are no parameters.

FitH, 1: top

Fit the page horizontally to the viewport. The top of the viewport is set to top.

FitV, 1: left

Fit the page vertically to the viewport. The left of the viewport is set to left.

FitR, 4: left, bottom, right, top

Fits the page to the provided rectangle with corners at (left, bottom) and (right, top).

FitB, 0

Fits the page to the bounding box of the page. There are no parameters.

FitBH, 1: top

Fits the page horizontally to the bounding box of the page. The top of the viewport is set to top.

FitBV, 1: left

Fits the page vertically to the bounding box of the page. The left of the viewport is set to left.

PDF::Writer#add_internal_link(label, x0, y0, x1, y1)

Creates an internal link in the document, ‘label’ is the name of an target point, and the other settings are the coordinates of the enclosing rectangle of the clickable area from (x0, y0) to (x1, y1).

Note that the destination markers are created using the #add_destination method described below. This the manual way of doing what <c:ilink> does.

PDF::Writer#add_link(uri, x0, y0, x1, y1)

Creates a clickable rectangular area from (x0, y0) to (x1, y1) within the current page of the document, which takes the user to the specified URI when clicked. This is the manual way of doing

what <c:alink> does.

```
pdf.add_link("http://rubyforge.org/projects/ruby-pdf/", pdf.left_margin,
100,
             pdf.left_margin + 400, 120)
pdf.text("http://rubyforge.org/projects/ruby-pdf/", :font_size => 16,
         :justification => :centre, :left => 50)
pdf.rectangle(50, 100, 400, 20).stroke
```

http://rubyforge.org/projects/ruby-pdf/

Text Rendering Style

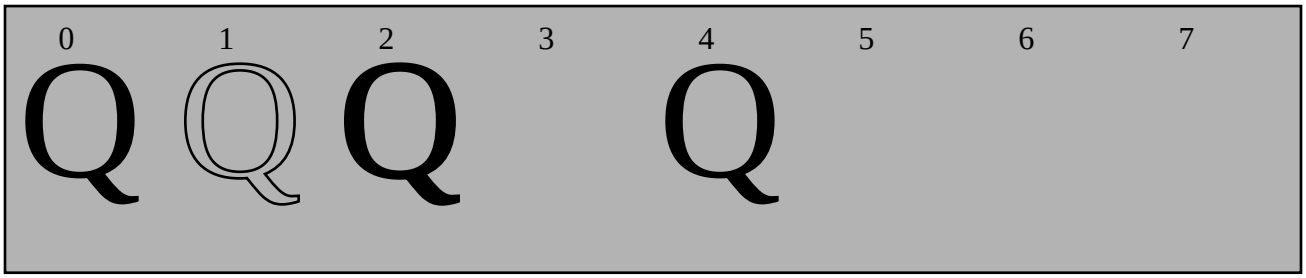
PDF normally renders text by filling the path described by the font in use. PDF::Writer provides (through PDF::Writer::Graphics) three methods to change the way that text is rendered.

#text_render_style(style)
#text_render_style!(style)
#text_render_style?

The first and second methods will set the text rendering style; the second forces the style to be set in the PDF document even if it's the same as the currently known text rendering style. The third method returns the current text rendering style.

The text rendering styles are integer values zero through seven. The meaning of "fill" and "stroke" are explained in-depth later. PDF::Writer does not support the use of clipping paths at this point.

- 0: (default) Fill the text. Uses #fill_color for text rendering colour.
- 1: Stroke the text.
- 2: Fill, then stroke the text.
- 3: Neither fill nor stroke the text (e.g., it is invisible).
- 4: Fill the text and add it to the clipping path.
- 5: Stroke the text and add it to the clipping path.
- 6: Fill, then stroke the text and add it to the clipping path.
- 7: Add the text to the clipping path.



5. Graphics in PDF Documents

Graphics primitives are provided by the module PDF::Writer::Graphics, included into PDF::Writer so that graphics primitives may be accessed from PDF document objects transparently. All drawing operations return the canvas so that drawing operations can be chained.

Drawing Paths

PDF graphics drawing operations are different than most drawing libraries. In most libraries, when a straight line is drawn between point A and point B, the line is drawn in the current colour, line thickness, and style. PDF drawing does not draw the line, but instead plots a path. Only when a path is painted will it be rendered in the document as a visible drawing.

As an illustration, with the following code:

```
pdf.move_to(70, 70).line_to(100, 100).line_to(40, 40).line_to(70, 70)
```

Nothing will be drawn until it is painted (with a stroke operation in this case):

```
pdf.stroke
```

The code for this can be simplified, too.

```
pdf.move_to(70, 70).line_to(100, 100).line_to(40, 40).close_stroke
```

Drawing paths may be arbitrarily long and may be comprised of several sub-paths. Paths are considered sub-paths when a new path is started. Only a paint operation finishes all drawing paths. Much of the precise behaviour will become clearer as the graphics operations are introduced over the next sections.

Drawing Colours and Styles

PDF documents keep track of two separate drawing colours at any given time: the stroke colour and the fill colour, used to render the respective paint operations (see Painting Paths). Stroke operations need not be rendered with solid lines; this is controlled by the Stroke Style.

Drawing Colours

Both stroke and fill colours should be set to instances of either Color::RGB or Color::CMYK, defined and documented in the color-tools package. This helper package is briefly documented in “ color-tools”.

#stroke_color(color)
#fill_color(color)
#stroke_color!(color = nil)
#fill_color!(color = nil)
#stroke_color?
#fill_color?

These six methods set and report the colour for stroke and fill operations, respectively. The first two

will only change the stroke/fill colour if it is different than the current stroke/fill colour. The second will force the current stroke/fill colour to be set or reset, unless a nil value is provided. The third returns the value of the current stroke/fill colour.

Stroke Style

Stroke styles must be instances of the PDF::Writer::StrokeStyle class.

PDF::Writer::StrokeStyle

This class represents how lines will be drawn with stroke operations. Line styles are defined by width, line cap style, join method, miter limit, and dash patterns.

PDF::Writer::StrokeStyle.new(width = 1, options = {})

Creates the new stroke style with the specified width and options. The options correspond to the attributes described below. If a block is provided, the stroke object is yielded to it.

PDF::Writer::StrokeStyle#width, PDF::Writer::StrokeStyle#width=

The thickness of the line in PDF units.

```
s1 = PDF::Writer::StrokeStyle.new(1)
s2 = PDF::Writer::StrokeStyle.new(2)
s3 = PDF::Writer::StrokeStyle.new(3)
s4 = PDF::Writer::StrokeStyle.new(4)
s5 = PDF::Writer::StrokeStyle.new(5)
```

PDF::Writer::StrokeStyle#cap, PDF::Writer::StrokeStyle#cap=

The type of cap to put on the ends of the line.

- :butt caps squre off the stroke at the endpoint of the path. There is no projection beyond the end of the path.
- :round caps draw a semicircular arc with a diameter equal to the line width is drawn around the endpoint and filled in.
- :square caps continue the stroke beyond the endpoint of the path for a distance equal to half the line width and is squared off.

```
s5.cap = :butt
```

```
s5.cap = :round
```

```
s5.cap = :square
```

If this is unspecified, the cap style is unchanged.

PDF::Writer::StrokeStyle#join, PDF::Writer::StrokeStyle#join=

How two lines join together.

- :miter joins indicate that the outer edges of the strokes for the two segments are extended until they meet at an angle, as in a picture frame. If the segments meet at too sharp an angle (as defined by the #miter_limit), a bevel join is used instead.

- :round joins draw an arc of a circle with a diameter equal to the line width is drawn around the point where the two segments meet, connecting the outer edges of the strokes for the two segments. This pie-slice shaped figure is filled in, producing a rounded corner.
- :bevel joins finish the two segments with butt caps and the the resulting notch beyond the ends of the segments is filled with a triangle, forming a flattened edge on the join.

```
s5.join = :miter
```

```
s5.join = :round
```

```
s5.join = :bevel
```

If this is unspecified, the join style is unchanged.

PDF::Writer::StrokeStyle#miter_limit, PDF::Writer::StrokeStyle#miter_limit=
When two line segments meet and <tt>:miter</tt> joins have been specified, the miter may extend far beyond the thickness of the line stroking the path. The #miter_limit imposes a maximum ratio miter length to line width at which point the join will be converted from a miter to a bevel. Adobe points out that the ratio is directly related to the angle between the segments in user space. With [p] representing the angle at which the segments meet:

```
miter_length / line_width == 1 / (sin ([p] / 2))
```

A miter limit of 1.414 converts miters to bevels for [p] less than 90 degrees, a limit of 2.0 converts them for [p] less than 60 degrees, and a limit of 10.0 converts them for [p] less than approximately 11.5 degrees.

PDF::Writer::StrokeStyle#dash, PDF::Writer::StrokeStyle#dash=
Controls the pattern of dashes and gaps used to stroke paths. This value must either be nil, or a hash with :pattern and :phase

The :pattern is an array of numbers specifying the lengths (in PDF userspace units) of alternating dashes and gaps. The array is processed cyclically, so that { :pattern => [3] } represents three units on, three units off, and { :pattern => [2, 1] } represents two units on, one unit off. These are shown against a solid line for comparison.

The :phase is an integer offset to the :pattern where the drawing of the stroke begins. This means that { :pattern => [3], :phase => 1 } represents be two units on and followed by a repeating pattern of three units off, three units on. A dash setting of { :pattern => [2, 1], :phase => 2 } repeats an off unit followed by two units on. These are shown against a solid line for comparison.

StrokeStyle#dash may be set to StrokeStyle::SOLID_LINE, returning drawing to a solid line.

Dashed lines wrap around curves and corners just as solid stroked lines do, with normal cap and join handling with no consideration of the dash pattern. A path with several subpaths treats each sub-path independently; the complete dash pattern is restarted at the beginning of each sub-path.

#stroke_style(style)
#stroke_style!(style = nil)
#stroke_style?(style)

These methods set and report the style for stroke operations. The first two will only change the stroke style if it is different than the current stroke style. The second will force the current stroke style to be set or reset, unless a nil value is provided. The third returns the value of the current stroke style.

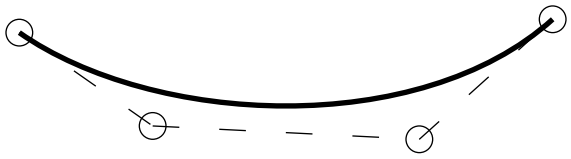
The default drawing style (a one unit solid line) can be set with the constant StrokeStyle::DEFAULT.

Cubic Bézier Curves

PDF drawings can only derive curved images (including ellipses and circles) through a concept known as cubic Bézier curves . These draw curved lines using four control points. (A demonstration applet can be found at the Princeton computer science department).

The distinctive feature of these curves is that the curve is guaranteed to be completely contained by the four sided polygon formed from the control points. The example below demonstrates a Bézier curve with its control points and three sides of the polygon marked. The curve is tangent to the line between the control points at either end of the curve.

```
pdf.curve(200, pdf.y + 40, 250, pdf.y + 5, 350, pdf.y, 400, pdf.y +
45).stroke
```



Drawing Path Primitives

These primitives control the drawing path that can later be painted. PDF documents have a single drawing point for drawing operations. The operators described here work with this drawing point and move the drawing point to a new position.

#move_to(x, y)

Moves the drawing point to (x, y). This will start a new sub-path.

#line_to(x, y)

Continues the current sub-path in a straight line from the drawing point to the new (x, y) coordiante. The drawing point is left at (x, y).

#curve_to(x0, y0, x1, y1, x2, y2)
This draws a Bézier curve with the drawing point (dpx, dpy) and (x2, y2) as the terminal points. (x0, y0) and (x1, y1) are the control points on the curve. The drawing pointer is set to (x2, y2) at the end of the curve.

#scurve_to(x0, y0, x1, y1)
This draws a Bézier spline with the drawing point (dpx, dpy) and (x1, y1) as the terminal points. (dpx, dpy) and (x0, y0) are the control points on the curve. The drawing pointer is set to (x1, y1) at the end of the curve.

#ecurve_to(x0, y0, x1, y1)
This draws a Bézier spline with the drawing point (dpx, dpy) and (x1, y1) as the terminal points. (x0, y0) and (x1, y1) are the control points on the curve. The drawing pointer is set to (x1, y1) at the end of the curve.

#rectangle(x, y, w, h = w)
Draws a rectangle from (x, y) to (x + w, y + h). The drawing pointer is set to (x + w, y + h) when complete. This is a basic PDF drawing primitive that continues the current drawing sub-path.

#close
Closes the current sub-path by appending a straight line segment from the drawing point to the starting point of the path. If the path is already closed, this is does nothing. This operator terminates the current sub-path.

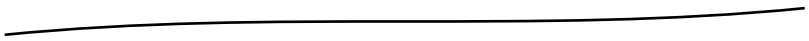
Shape Operators
These operators build on the drawing path primitives to provide complete shapes for drawing. These will always generate new sub-paths.

#line(x0, y0, x1, y1)
Draws a straight line from (x0, y0) to (x1, y1) and leaves the drawing pointer at (x1, y1).

```
pdf.line(100, pdf.y - 5, 400, pdf.y + 5)
```

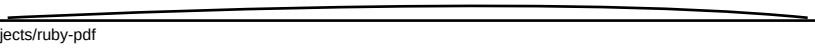
#curve(x0, y0, x1, y1, x2, y2, x3, y3)
This draws a Bézier curve with (x0, x1) and (x3, y3) as the terminal points. (x1, y1) and (x2, y2) are the control points on the curve. The drawing pointer is set to (x3, y3) at the end of the curve.

```
pdf.curve(100, pdf.y - 5, 200, pdf.y + 5, 300, pdf.y - 5, 400, pdf.y + 5
```



#scurve(x0, y0, x1, y1, x2, y2)
This draws a Bézier spline with (x0, y0) and (x2, y2) as the terminal points. (x0, y0) and (x1, y1) are the control points on the curve. The drawing pointer is set to (x2, y2) at the end of the curve.

```
pdf.scurve(100, pdf.y - 5, 300, pdf.y + 5, 400, pdf.y - 5)
```



#ecurve(x0, y0, x1, y1, x2, y2)

This draws a Bézier spline with (x0, y0) and (x2, y2) as the terminal points. (x1, y1) and (x2, y2) are the control points on the curve. The drawing pointer is set to (x2, y2) at the end of the curve.

```
pdf.ecurve(100, pdf.y - 5, 200, pdf.y + 5, 400, pdf.y - 5)
```


#circle_at(x, y, r)

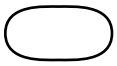
Draws a circle of radius r with centre point at (x, y). The drawing pointer is moved to (x, y) when finished.

```
pdf.circle_at(250, pdf.y, 10)
```

#ellipse_at(x, y, r1, r2 = r1)

Draws an ellipse of horizontal radius r1 and vertical radius r2 with centre point at (x, y). The drawing pointer is moved to (x, y) when finished.

```
pdf.ellipse_at(250, pdf.y, 20, 10)
```



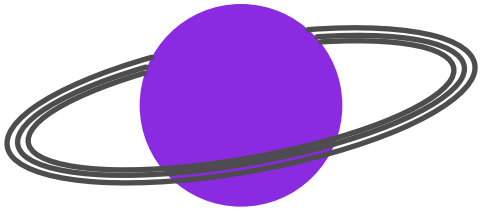
#ellipse2_at(x, y, r1, r2 = r1, start = 0, stop = 360, segments = 8)

Draws an ellipse with centre point at <tt>(x, y)</tt>. The horizontal radius is r1 and the vertical radius is r2. A partial ellipse may be drawn by specifying the starting and finishing angles in degrees. The ellipse is drawn in segments composed of cubic Bézier curves. It is not recommended that this be set to a value smalelr than 4, and any value less than 2 will be treated as 2. The drawing pointer is moved to (x, y) when finished. This ellipse is more “natural” at smaller sizes.

```
pdf.ellipse2_at(250, pdf.y, 20, 10)
```

```
pdf.ellipse2_at(250, pdf.y, 20, 10, 135, -45)
```

```
ss = PDF::Writer::StrokeStyle.new(2)
ss.cap = :round
pdf.stroke_style  ss
pdf.fill_color! Color::RGB::BlueViolet
pdf.translate_axis(300, pdf.y + 25)
pdf.circle_at(0, 0, 38).fill
pdf.rotate_axis(10)
pdf.stroke_color! Color::RGB::Grey30
pdf.ellipse2_at(0, 0, 81, 20, 65.5, -245.5).stroke
pdf.ellipse2_at(0, 0, 85, 22, 67.5, -247.5).stroke
pdf.ellipse2_at(0, 0, 89, 25, 70.5, -250.5).stroke
pdf.restore_state
```



#segment_at(x, y, r1, r2 = r1, start = 0, stop = 359.99, segments = 8)

Draws an ellipse segment. This is a closed partial ellipse with lines from the starting point to the centre point and the ending point and the centre point. The drawing pointer is moved to (x, y) when finished.

```
pdf.segment_at(250, pdf.y, 20, 10, 45, -45)
```

#polygon(points)

Draws a polygon using points. This is an array of PDF::Writer::PolygonPoint objects or an array that can be converted to an array of PolygonPoint objects. A PolygonPoint is a simple object that contains (x, y) coordinates for the point and the way that the line will be connected to the previous point. The connnector may be either :curve, :scurve, :ecurve, or :line.

The connector is ignored on the first point provided to the polygon; that is the starting point.

The drawing point is left at the last (x, y) point position.

:curve

A :curve connector indicates that there are three points following this point, to be provided as parameters to #curve_to.

```
points = [ [ 10, 10 ],          # starting point
           [ 20, 20, :curve ],  # first control point
           [ 30, 30 ],          # second control point
           [ 40, 40 ],          # ending point of the curve.
         ]
```

:scurve

An :scurve connector indicates that there are two points following this point, to be provided as parameters to #scurve_to.

```
points = [ [ 10, 10 ],          # starting point and first control point
           [ 20, 20, :scurve ], # second control point
           [ 30, 30 ],          # ending point of the curve.
         ]
```

:ecurve

An :ecurve connector indicates that there are two points following this point, to be provided as parameters to #ecurve_to.

```
points = [ [ 10, 10 ],          # starting point
           [ 20, 20, :ecurve ], # first control point
           [ 30, 30 ],          # ending point and second control point
         ]
```

:line

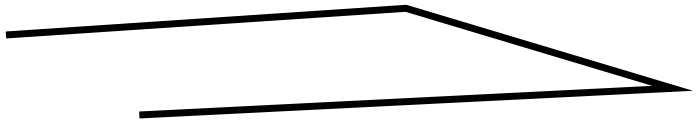
A :line connector draws a line between the previous point and this point.

```
points = [ [ 10, 10 ],          # starting point
           [ 20, 20, :line ],   # ending point
         ]
```

Examples

All of these examples use :line connectors.

```
pdata = [[200, 10], [400, 20], [300, 50], [150, 40]]
pdf.polygon(pdata)
```



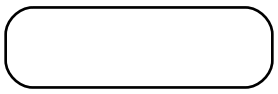
```
pdf.polygon(pdata).fill
```

```
pdf.fill_color(Color::RGB.from_fraction(0.9, 0.8, 0.7))
pdf.polygon(pdata).fill
```


#rounded_rectangle(x, y, w, h, r)

Draws a rectangle with rounded corners. The drawing point is left at the (x + w, y + h). The radius should be smaller than the width and height.

```
pdf.rounded_rectangle(100, pdf.y, 300, 30, 5)
```



#star(cx, cy, length, rays = 5)

Draws a star centered at (cx, cy) with the specified number of rays, where each ray is length units long. Stars with an odd number of rays should have the top ray pointing toward the top of the document. If the number of rays is less than four, it will be converted to four.

```
pdf.star(166, pdf.y, 20, 4).stroke
pdf.star(232, pdf.y, 20, 5).stroke
pdf.star(298, pdf.y, 20, 6).stroke
pdf.star(364, pdf.y, 20, 7).stroke
pdf.star(430, pdf.y, 20, 8).stroke
```

Painting Paths

Drawing paths become visible in PDF documents when they are painted. In painting a path, the path may be stroked or filled.

Stroking Operations

Painting a path with a stroke draws a line along the current path. The line follows each segment in the path, centered on the segment with sides parallel to it. Each segment of the line uses the current line width and dash pattern. Lines that meet use the current join style; ends of lines unconnected to other lines use the current line cap style.

When two unconnected segments meet or intersect in the same space, they will not be drawn as a closed segment; an explicit #close operator is recommended in this case. If paths are manually closed, according to the Adobe PDF Reference Manual version 1.6, the result may be a “messy corner, because line caps are applied instead of a line join.”

There is the special case of single-point paths; when a path is either a closed path that is exactly one point in size or of two or more points at the same coordinates, the #stroke operator only paints the path if :round line caps are specified, producing a filled circle centered at the single point. This is considered a “degenerate” sub-path. An open degenerate sub-path is not drawn.

Filling Operations

Fill paint operations colour the entire region enclosed by the current path with the current fill colour. If there are several disconnected subpaths, open subpaths are implicitly closed and the overall enclosed area is painted—the space is “joined” before painting.

Degenerate subpaths (the same as described above) will result in the painting of the single device pixel lying under that point—this is device dependent and not useful. An open degenerate sub-path is not drawn.

Filling simple paths is intuitively clear: parts of the document enclosed in the path will be painted in the current fill colour. Complex paths—those that intersect themselves or have one or more sub-paths enclosing other sub-paths—require additional considerations for knowing what portions of the canvas are inside a path. PDF offers two different rules: nonzero winding number and the even-odd.

Nonzero Winding Number Fill Rule

This is the standard fill behaviour. In this rule, a given point is inside a path by conceptually drawing a line from the point to beyond the outer edge of the path. Using an initial value of zero, at every place where the path crosses the ray from left to right, one will be added to the value; where it crosses from right to left, one will be subtracted from the value. If the resulting value is zero, the point is outside of the path. In all other cases, the path is inside. This can be expressed in Ruby as:

```
crossings.inject(0) { |ii, xd| xd.left_right? ? ii + 1 : ii - 1 } != 0
```

For the purposes of applying this rule, if a ray concides with or is tangent to a path segment, a different ray is chosen. Consider the two six-pointed stars below. The first is drawn so that all paths head in the same direction:

```
pdf.move_to(x0, y1).line_to(x2, y1).line_to(x1, y3).close
pdf.move_to(x1, y0).line_to(x2, y2).line_to(x0, y2).close
```

The second is drawn differently:

```
pdf.move_to(x0, y1).line_to(x2, y1).line_to(x1, y3).close
pdf.move_to(x0, y2).line_to(x2, y2).line_to(x1, y0).close
```

In the first, the figure is solid. In the second, because of the nonzero winding number fill rule, it is hollow.

Even-Odd Fill Rule

The even-odd fill rule uses the same imaginary ray concept as the nonzero winding number. Instead of depending on the direction of the intersections of path with the ray, it depends on the total number of intersections. If that total is odd, the point is inside the path for painting purposes. If the total is even, the point is outside the path. Shown below are the same two figures used in the nonzero winding number rule, but this time they are filled using the even-odd rule. They appear the same.

Fill Methods

#stroke

Terminate the path object and draw it with the current stroke style.

#close_stroke

Close the current path by appending a straight line segment from the drawing point to the starting point of the path and then stroke it. This is the same as “pdf.close.stroke”.

#fill(rule = nil)

Fills the path according to the rules noted above, using the nonzero winding number rule unless the rule parameter has the value :even_odd. Uses the current fill colour.

#close_fill(rule = nil)

Close the current path by appending a straight line segment from the drawing point to the starting point of the path and then fill it. This is the same as “pdf.close.fill”.

#fill_stroke(rule = nil)

Fills and strokes the path. This is the same as constructing two identical path objects and calling #fill on the first and #stroke on the second. Paths filled and stroked in this manner are treated as if they were one object for PDF transparency purposes (the PDF transparency model is not yet supported by PDF::Writer).

#close_fill_stroke(rule = nil)
Closes (as per #close), fills (as per #fill), and strokes (as per #stroke) the drawing path. As with #fill_stroke, it constructs a single object that acts like two paths.

Images

PDF::Writer supports the insertion of images into documents. These images must be either JPEG (either RGB or CMYK colour spaces) or PNG format; other format images must be converted to JPEG or PNG before they can be used in a document.

Images are inserted into PDF documents as one pixel per PDF userspace unit, making a 320×240 image approximately 4½”×3¼” (113mm×85mm) in size.

#add_image(image, x, y, width = nil, height = nil, image_info = nil, link = nil)

Adds an image to the PDF document. The image must be a String that represents the binary contents of a PNG or JPEG file or it must be a PDF image that has previously been inserted by one of the image insertion methods (including this one).

The loaded image will be placed with its lower left-hand corner at (x, y) on the current page. The size of the image will be determined by the width and height parameters. If neither width nor height are provided, the image will be inserted at its “natural” size.

If width is specified, but height is not specified, then the image will be scaled according to the width of the image. If height is specified, but width is not, the image will be scaled according to the height of the image. In both cases, the aspect ratio—the ratio of width to height—will be maintained. If both width and height are specified, the set values will be used.

If the width and/or height values are negative, the image will be flipped horizontally or vertically, as appropriate.

The image_info parameter must either be unspecified, nil, or a valid instance of PDF::Writer::Graphics::ImageInfo (see “ PDF::Writer::Graphics::ImageInfo”).

The link parameter allows the image to be made as a clickable link to internal or external targets. It must be specified as a hash with two options:

- :type is the type of the link, either :internal or :external. This affects the interpretation of :target.
- :target is the destination of the link. For :type => :internal links, this is an internal destination. For :type => :external links, this is an URI. See #add_internal_link and #add_link for more information.

The image object will be returned so that it can be used again in the future.

#add_image_from_file(image, x, y, width = nil, height = nil, link = nil)

Adds an image to the PDF document from a file. The image parameter may be an IO-like object (it must respond to #read and return the full image data in a single read) or a filename. If open-uri is

active, the filename may also be an URI to a remote image.

In all other ways, it behaves just like #add_image.

#image(image, options = {})

This third method for inserting images into a PDF document holds much the same relationship to #add_image and #add_image_from_file as #text holds to the text insertion methods #add_text and #add_text_wrap. This method will insert an image in the document positioned vertically relative to the current internal writing pointer. The horizontal position and image size are controlled by the options passed to the method. If the image is taller than the available space on the page and the image is not to be resized to the available space (see below), a new page will be created and the image will be inserted there.

:pad

The image will be inserted into the document with this number of blank userspace units on all sides. The default is 5 units.

:width

The desired width of the image in userspace units. The image will be resized to this width using the aspect ratio of the image. The image will not be allowed to be larger than the amount of space available for the image as controlled by the document margins and the padding. So, if a page is 400 units wide with 50 unit margins, that leaves an available size of 300 units. With the default padding of 5 units, the inserted picture will be no larger than 290 units wide, even if :width is set larger.

:resize

How to automatically resize the image. This may be either :width, :full, or a numeric value.

- :resize => :width will resize the image to be as wide as the writing area. The image height will be resized in accordance with the aspect ratio of the image. If the resized image is taller than will fit in the remaining space on the page, a new page will be created and the image will be inserted at the top of the next page.
- :resize => :full will resize the image to be as wide as the writing area. The image height will be resized in accordinace with the aspect ratio of the image. If the resized image is taller than will fit in the remaining space on the page, the image will be resized again to be as tall as the remaining space. The width will be correspondingly shrunk.
- :resize => number will resize the image by a factor of the provided number. :resize => 2 will double the size of the image and :resize > 0.5 will halve the size of the image. If the resized image is wider than the maximum width, it will be resized again to be no wider than that, with a proportional height.

This option is incompatible with the :width option. The behaviour when both are specified is undefined.

:justification

The horizontal placement of the image. It may be :center (the centre of the image is placed in the centre of the margins and padding space), :right (the right edge of the image is flush with the right margin and padding space), or :left (the left edge of the image is flush with the left margin and padding space). The default justification is :left.

:border
The image may optionally be drawn with a border. If this parameter is true, the default border will be drawn, which is a 50% grey solid line border. The parameter may also be a hash with two values.

- :color specifies the colour of the border.
- :style specifies the stroke style of the border.

:link
The image may be made a clickable link to internal or external targets. This option must be specified as a hash with two options:

- :type is the type of the link, either :internal or :external. This affects the interpretation of :target.
- :target is the destination of the link. For :type => :internal links, this is an internal destination. For :type => :external links, this is an URI. See #add_internal_link and #add_link for more information.

Limitations
Because #text and #image comprise a simple layout engine, the limitations of these methods must be understood. The image will be inserted and text will be inserted after it. There is no option to place the image and then wrap text around it as one might with OpenOffice or a proper destkop publishing system. The one example of anything similar in this manual (page 4) was done manually.

Coordinate Axis Transformations
As described earlier in “ PDF Coordinate Space,” the standard PDF coordinate space has the base coordinate (0, 0) in the lower left-hand corner of the canvas. PDF::Writer’s layout engine depends on this behaviour, but the axis can be transformed in several ways. It can be translated, rotated, scaled, and skewed.

Axis transformations are cumulative (matrix multiplicative; see the description of #transform_matrix for a full explanation) within a graphics state stack level (see the next section). It is recommended that axis transformations are performed within a #save_state/#restore_state combination.

translate_axis(x, y)
Translates the coordinate space so that (x, y) under the old coordinate space is now (0, 0). The direction of the old coordinate space is unchanged (y values are increasing). The example code below will make the middle of the page the (0, 0) position.

```
pdf.translate_axis(pdf.margin_x_middle, pdf.margin_y_middle)
```

rotate_axis(angle)
The entire coordinate space is rotated by the specified angle in a counter-clockwise direction.

NOTE: As of PDF::Writer 1.1.0, angle rotations are now counter-clockwise, not clockwise as in earlier versions. This is a necessary incompatible change to make transformations more compatible with other vector graphics systems.

scale_axis(x = 1, y = 1)

Changes the size of PDF userspace units. This changes the scale of all operations in PDF documents. With #scale_axis(0.5, 1), the x-axis is modified so that PDF userspace units are 1/144” in width, instead of the normal 1/72” width. Note that PDF::Writer takes no notice of scale changes for purposes of margins.

If the scale value is negative, the coordinate axis is reversed and graphics or text will be drawn in reverse.

skew_axis(xangle = 0, yangle = 0)

Rotates the x and y axes independently of one another, creating a skewed appearance.

transform_matrix(a, b, c, d, e, f)

Transforms the coordinate axis with the transformation vector interpreted as a coordinate transformation matrix:

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

The matrix transforms the new coordinates (x1, y1) to the old coordinates (x0, y0) through the dot product:

$$\begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix}$$

In practice, the six variables (a—f) be represented as a six-element vector: [a b c d e f].

- Axis translation uses [1 0 0 1 x y] where x and y are the new (0,0) coordinates in the old axis system.
- Scaling uses [sx 0 0 sy 0 0] where sx and sy are the scaling factors.
- Rotation uses [cos(a) sin(a) -sin(a) cos(a) 0 0] where a is the angle, measured in radians.
- X axis skewing uses [1 0 tan(a) 1 0 0] where a is the angle, measured in radians.
- Y axis skewing uses [1 tan(a) 0 1 0 0] where a is the angle, measured in radians.

Graphics State

PDF allows for multiple independent levels of graphics state to be saved in a last-in first-out stack. Graphics states may not remain open over page boundaries, and must be balanced; the use of the two methods below will ensure that this is managed automatically.

PDF::Writer#save_state

This operation saves the current graphics state on a stack. This stack keeps various parameters about the graphics state.

- The stroke style (line width, line cap style, line join style, miter limit, and dash pattern), page 31.
- The coordinate axis transformation matrix, page 42.
- The colour rendering intent (not yet supported by PDF::Writer).
- The flatness tolerance (not yet supported by PDF::Writer).
- Extended graphics state dictionaries (not yet supported by PDF::Writer).

PDF::Writer#restore_state

Restores the current graphics state.

PDF::Writer::Graphics::ImageInfo

This class obtains metadata information about an image. It is a modified version of ImageSize by Keisuke Minami, which can be found at http://www.rubycgi.org/tools/index.en.htm. It is available under the standard PDF::Writer licence but this class is also available under the GNU General Public Licence, version 2 or later.

It supports GIF (GIF87a and GIF89a), PNG (Portable Network Graphics), JPEG, BMP (Windows or OS/2 Bitmaps), PPM, PBM, PGM, TIFF, XBM (X Bitmap), XPM (X Pixmap), PSD (PhotoShop), PCX, and SWF (Flash) image formats.

ImageInfo.new(data, format = nil)

Creates the ImageInfo object from the provided image data. If format is specified, the image will be treated as the desired format. Otherwise, it will be automatically discovered (this is preferred).

ImageInfo#format

Returns the format of the image.

ImageInfo#height
ImageInfo#width

Returns the height and width of the image.

ImageInfo#bits

Returns the colour bit depth of the image, if supported.

ImageInfo#channels

Returns the number of colour channels, if supported.

ImageInfo#info

Returns other information the image may know about itself.

6. PDF::Writer Document Operations

There are several operations for manipulating the generated document as a whole or metadata about the document.

Whole Document Operations

These operations act on, or return information about, the entire document.

PDF::Writer#size

Returns the number of PDF objects in the document.

PDF::Writer#compressed, PDF::Writer#compressed=, PDF::Writer#compressed?

Indicates or sets document compression. If the value is true, the document will be compressed on writing using the “deflate” compression method. If the ‘zlib’ library cannot be loaded, compression settings will be ignored.

NOTE: This value should be set as early as possible during the document generation, or only some sections of the PDF will be compressed.

PDF::Writer#media_box(x0, y0, x1, y1) PDF::Writer#trim_box(x0, y0, x1, y1)

The #trim_box and the #media_box are both expressed in default (unscaled) user space units. The #media_box defines the boundaries of the physical medium on which the page is intended to be displayed or printed. As of this version, PDF::writer supports only one media box for the entire document. The document’s #trim_box is the intended dimensions of the finished page after trimming.

Document Metadata

These operations modify or provide information on, the metadata of the document.

PDF::Writer#version

The version of PDF to which this document conforms. Should be one of ‘1.3’ (PDF_VERSION_13), ‘1.4’ (PDF_VERSION_14), ‘1.5’ (PDF_VERSION_15), or ‘1.6’ (PDF_VERSION_16). This value is set during document creation.

PDF::Writer#info

The PDF document metadata. This is the document’s PDF::Writer::Object::Info object, and the metadata are attributes on this object. All metadata values are optional.

Info#title, Info#title=

The document’s title.

Info#author, Info#author=

The name of the person who created the document.

Info#subject, Info#subject=
The subject of the document.

Info#keywords, Info#keywords=
Keywords associated with the document.

Info#creator, Info#creator=
If the document was converted to PDF from another format, the name of the application that created the original document from which it was converted. This defaults to the name of the script that uses PDF::Writer.

Info#producer, Info#producer=
If the document was converted to PDF from another format, the name of the application (for example, Acrobat Distiller) that converted it to PDF. This defaults to "PDF::Writer for Ruby"

Info#creationdate, Info#creationdate=
The date and time the document was created, as a Ruby Time object. This defaults to the current date and time.

Info#moddate, Info#moddate=
The date and time the document was most recently modified, as a Ruby Time object. This value is required if a PieceInfo object is in the document catalog—which is not currently supported by PDF::Writer.

Info#trapped, Info#trapped=
A name object indicating whether the document has been modified to include trapping information. This is not currently supported by PDF::Writer.

PDF::Writer#viewer_preferences(label, value = 0)
This will set values that indicate to the viewer how it should display the itself with respect to this document.

The viewer preferences to be set. This may be passed as a single pair value or a hash of names and values.

```
pdf.viewer_preferences("HideToolbar", true)
pdf.viewer_preferences({ "HideToolbar" => true, "HideMenubar" => true})
```

HideToolbar
A flag specifying whether to hide the viewer application tool bars when the document is active. Default value: false.

HideMenubar
A flag specifying whether to hide the viewer application menu bar when the document is active. Default value: false.

HideWindowUI
A flag specifying whether to hide user interface elements in the document window (such as scroll bars and navigation controls), leaving only the document contents displayed. Default value: false.

FitWindow
A flag specifying whether to resize the document window to fit the size of the first displayed page. Default value: false.

CenterWindow
A flag specifying whether to position the document window in the center of the screen. Default value: false.

NonFullScreenPageMode
The document page mode, specifying how to display the document on exiting full-screen mode. This entry is meaningful only if the value of the PageMode entry in the catalog dictionary is FullScreen; it is ignored otherwise. Default value: UseNone.

Permitted names are:
- UseNone: Neither document outline nor thumbnail images visible
- UseOutlines: Document outline visible
- UseThumbs: Thumbnail images visible

Direction
The predominant reading order for text: L2R Left to right R2L Right to left (including vertical writing systems such as Chinese, Japanese, and Korean) This entry has no direct effect on the document contents or page numbering, but can be used to determine the relative positioning of pages when displayed side by side or printed n-up. Default value: L2R.

PDF::Writer#open_here(style, *params)
PDF::Writer#open_at(page, style, *params)
Specify the destination object where the document should open when it first starts. +style+ must be one of the values detailed for #add_destination. The value of +style+ affects the interpretation of +params+. PDF::Writer#open_here uses the current page as the starting location.

PDF::Writer#encrypt(user_pass = nil, owner_pass = nil, permissions = [])
Encrypts the document using Adobe RC4. This will set the user and owner passwords that will be used to access the document and set the permissions the user has with the document. The passwords are limited to 32 characters.

The permissions provided are an array of Symbols, allowing identified users to perform particular actions:
- :print allows printing.
- :modify allows the modification of text or objects.
- :copy allows the ability to copy text or objects.
- :add allows the ability to add text or objects.

Encrypting the document without passwords or permissions prevents the user from using copy and paste operations or printing the document with no passwords.

Setting either of the passwords will mean that the user will have to enter a password to open the

document. If the owner password is entered when the document is opened then the user will be able to print etc. If the two passwords are set to be the same (or the owner password is left blank) then there is noo owner password, and the document cannot be opened in the accesible mode.

The following example sets an owner password, a user password, and allows printing and copy/paste operations.

```
pdf.encrypt('trees', 'frogs', [ :copy, :print ])
```

PDF::Writer#save_as(filename)

Saves the PDF document as a file to disk.

7. Tables in PDF Documents

It is very common for information to be displayed in tabular form. PDF documents have no inherent concept of tabular data, leaving it up to the generating application to create tables (or other structured forms). The PDF::Writer distribution solves this with the PDF::SimpleTable class.

PDF::SimpleTable

PDF::SimpleTable is so called because it has a relatively simple table model, particularly as compared to the more complex (and more flexible) tables present in HTML or XHTML.

At its simplest, PDF::SimpleTable will take its data (an array, where each row is a hash containing the columns of data) and its column definitions, and render the table on the PDF document.

Everything else is configuration for different ways of displaying the data.

Tables will start drawing from the current writing point and will continue drawing until all of the data has been presented. By default, outer borders will be drawn; alternate rows will be shaded, and the table will wrap across pages, displaying header rows at the top of each page. The rendering method returns the Y position of the writing pointer after the table has been added to the document.

The following sections will cover the creation of the table, structural elements (required and optional), text options, border and shading options, positioning options, and pagination opitons.

PDF::SimpleTable.new

This creates a SimpleTable with default values. It will yield the created table if a block is provided.

```
require 'pdf/simpletable'
tab = PDF::SimpleTable.new
PDF::SimpleTable.new do |table|
  # ...
end
```

Structural Elements

A SimpleTable cannot render itself if it does not know what columns are to be displayed or if it does not have any data. Thus, #data and #column_order must be defined before the table rendering code will work. Optionally, options can be specified for the display of the columns through #columns.

#data, #data= (required)

This array contains the data to be displayed, where the elements of the array are the rows. Each row is a hash. They keys of the hash are the identifiers of the columns; the corresponding values are the values of each cell in the table.

```
tab.data = [
  { "row" => 1, "name" => "Gandalf",  "race" => "Human" },
  { "row" => 2, "name" => "Bilbo",    "race" => "Hobbit",
      "url" => "http://www.glyphweb.com/arda/b/bilbobaggins.html", }
  { "row" => 3, "name" => 'Frodo',    "race" => "Hobbit" },
  { "row" => 4, "name" => 'Saruman',  "race" => "Human",
      "url" => "http://www.lord-of-the-rings.org/books/saruman.html", }
  { "row" => 5, "name" => 'Sauron',   "race" => "???" }
]
```

#column_order, #column_order= (required)

This array serves a dual purpose. It defines both the columns that will be displayed and the order in which they will be displayed. Thus, with a column order like the one following, the Lord of the Rings characters will be displayed in a three-column table.

```
tab.column_order = [ "row", "name", "race" ]
```

Note that the "url" column will not be displayed. But we can do other things as well, like eliminate the "row" column entirely and reverse the order of the remaining two columns.

```
tab.column_order = [ "race", "name" ]
```

#columns, #columns=

This is a hash of PDF::SimpleTable::Column objects that defines various formatting options for the column data cells and, optionally, the column heading cells. If this is empty, or is missing entries for displayed columns, a Column object will be created using only the column's name.

PDF::SimpleTable::Column

Defines formatting options for an entire column, including optionally the column heading cells.

PDF::SimpleTable::Column.new(name)

Creates a column formatting option for the column known by name. Yields the created column if a block is passed to Column.new.

```
race_col = PDF::SimpleTable::Column.new("race")
```

Column#heading, Column#heading=

The heading row of the column; if headings are displayed, this row and value will be repeated when a table crosses a page boundary. If this is not an instance of PDF::SimpleTable::Column::Heading, it will be converted to one. If this value is unset, Column#name will be used for display in the heading cells.

```
race_col.heading = "Race"
```

Column#name

The name of the column. This value will be used if Column#heading is unset and headings are to be displayed.

Column#width, Column#width=

The width of the column in PDF userspace units. If this value is set, the column will be exactly this wide and the text in the column's cells may be wrapped to fit.

```
race_col.width = 90
```

Column#link_name, Column#link_name=

The data key name that will be used to provide an external hyperlink for values in this column. Internal hyperlinks are not automatically supported.

```
race_col.link_name = "uri"
```

Column#justification, Column#justification=

The justification of the data within the column. This applies to both headings and normal cell data.

As with normal text formatting, the values may be :left (the default), :right, :center, or :full.

```
race_col.justification = :center
```

PDF::SimpleTable::Column::Heading

Headings for columns may have more specific formatting options applied to them.

PDF::SimpleTable::Column::Heading.new(title = nil)
Creates the Heading object with an optional title to be displayed when headings are to be displayed.

```
race_col.heading = PDF::SimpleTable::Column::Heading.new("Race")
```

Heading#title, Heading#title=
The heading title, to be displayed when headings are displayed on the table. If this is empty, or there is no Heading defined for the column, the column's name will be used to display in the heading cells when needed.

```
race_col.heading.title = "Species"
```

Heading#justification, Heading#justification=
The justification of the heading cell for the column. As with normal text formatting, the values may be :left (the default), :right, :center, or :full.

```
race_col.heading.justification = :center
```

Heading#bold, Heading#bold=
Sets this table cell heading to be rendered bold. Independent of the table-wide #bold_headings setting.

Text Options

These options control the display of the text in and around the table.

SimpleTable#title, SimpleTable#title=

The title to be put on top of the table. This is only shown once, not on every page. Strictly speaking, it is not part of the table, but it is closely associated with the table.

SimpleTable#show_headings, SimpleTable#show_headings=

Displays the headings for the table if true, the default.

SimpleTable#font_size, SimpleTable#font_size=

The font size of the data cells, in points. Defaults to 10 points.

SimpleTable#heading_font_size, SimpleTable#heading_font_size=

The font size of the heading cells, in points. Defaults to 12 points.

SimpleTable#title_font_size, SimpleTable#title_font_size=

The font size of the #title, in points. Defaults to 12 points.

SimpleTable#title_color, SimpleTable#title_color=
The text colour of the #title. Defaults to Color::RGB::Black.

SimpleTable#heading_color, SimpleTable#heading_color=
The text colour of the heading. Defaults to Color::RGB::Black.

SimpleTable#text_color, SimpleTable#text_color=
The text colour of the body cells. Defaults to Color::RGB::Black.

SimpleTable#bold_headings, SimpleTable#bold_headings=
Makes the heading text bold if true. The default is false.

Border and Shading Options
These options control the appearance of the borders (if shown) and row shading (if enabled).

SimpleTable#show_lines, SimpleTable#show_lines=
Controls whether and how lines should be shown on the table.

- :none will display no lines on the table.
- :outer will display the outer lines of the table (the table's "frame"). This is the default.
- :inner will display the inner lines of the table (between cells only).
- :all will display the inner and outer lines of the table.

SimpleTable#shade_rows, SimpleTable#shade_rows=
Controls whether rows should be striped (that is, alternating one colour or the other).

- :none will shade no rows; all rows are the background colour.
- :shaded will shade alternate lines. Half of the rows will be the background colour; the rest of the rows will be shaded with #shade_color. This is the default.
- :striped will shade alternate lines with two colours. Half of the lines will be #shade_color; the other half will be #shade_color2.

Shade colours should be chosen carefully; this only controls the background colour of the row (the entire row), not the foreground colour.

SimpleTable#shade_color, SimpleTable#shade_color=
SimpleTable#shade_color2, SimpleTable#shade_color2=

The main and alternate shading colours. #shade_color defaults to Color::RGB::Grey80; #shade_color2 defaults to Color::RGB::Grey70.

SimpleTable#shade_headings, SimpleTable#shade_headings=
SimpleTable#shade_heading_color, SimpleTable#shade_heading_color=
If #shade_headings is true, heading rows will be shaded with the value of #shade_heading_color (Color::RGB::Grey90 by default). Headings are not shaded by default.

SimpleTable#line_color, SimpleTable#line_color=
The colour of table lines, defaultting to Color::RGB::Black.

SimpleTable#inner_line_style, SimpleTable#inner_line_style=
SimpleTable#outer_line_style, SimpleTable#outer_line_style=

Defines the inner and outer line styles. The default style for both is PDF::Writer::StrokeStyle::DEFAULT.

Positioning Options

These options control how and where the table, or some elements of a table, will be positioned. All measurements here are in normal PDF userspace units.

SimpleTable#title_gap, SimpleTable#title_gap=

This is the gap between the #title and the table. Default: 5 units.

SimpleTable#position, SimpleTable#position=
SimpleTable#orientation, SimpleTable#orientation=

SimpleTable uses two values to determine how the table will be placed on the page. The #position is the point from which the table will be placed, and it will be placed relative to that #position by the #orientation. It could be said that #position is the "origin" of the table and #orientation is how that origin is interpreted.

Position Values

- :left sets the origin of the table at the left margin.
- :right sets the origin of the table at the right margin.
- :center puts the origin of the table between the margins (default).
- A numeric offset value makes the origin of the table an offset position from the left of the page.

Orientation Values

- :left draws the table to the left of #position. The right border is at #position.
- :right draws the table to the right of #position. The left border is at #position.
- :center draws the centre of the table at #position (default). The table will extend both left and right of #position.
- A numeric offset draws the table to the right of #position + offset. The left border is at #position + offset.

Combinations

```
tab.position    = :left   # The table ...
tab.orientation = :left   # extends LEFT from the left margin.
tab.orientation = :right  # extends RIGHT from the left margin.
tab.orientation = :center # is centered on the left margin.
tab.orientation = 35      # extends RIGHT from the left margin + 35.
tab.orientation = -35     # extends RIGHT from the left margin - 35.
tab.position = :right     # The table ...
tab.orientation = :left   # extends LEFT from the right margin.
tab.orientation = :right  # extends RIGHT from the right margin.
tab.orientation = :center # is centered on the right margin.
tab.orientation = 35      # extends RIGHT from the right margin + 35.
tab.orientation = -35     # extends RIGHT from the right margin - 35.
tab.position = :center    # The table...
tab.orientation = :left   # extends LEFT from the margin middle.
tab.orientation = :right  # extends RIGHT from the margin middle.
tab.orientation = :center # is centered between margins. This is the
default.
```

```
tab.orientation = 35      # extends RIGHT from the margin middle + 35.
tab.orientation = -35     # extends RIGHT from the margin middle - 35.
tab.position = 35         # The table...
tab.orientation = :left   # extends LEFT from 35.
tab.orientation = :right  # extends RIGHT from 35.
tab.orientation = :center # is centered on 35.
tab.orientation = 35      # extends RIGHT from 70.
tab.orientation = -35     # extends RIGHT from 0.
tab.position = -35        # The table...
tab.orientation = :left   # extends LEFT from -35.
tab.orientation = :right  # extends RIGHT from -35.
tab.orientation = :center # is centered on -35.
tab.orientation = 35      # extends RIGHT from 0.
tab.orientation = -35     # extends RIGHT from -70.
```

SimpleTable#width, SimpleTable#width=
SimpleTable#maximum_width, SimpleTable#maximum_width=

There are two ways to control the width of the table. Setting the #width of the table forces the table to be that wide, regardless of how much data is in the table. If the table is smaller than this width, the difference is added proportionately to each unsized column to make the table exactly this wide. If the table is larger than this width, unsized columns are reduced in size to fit the available space. This may cause some data to be wrapped in the cells. The default value is zero (0); this will automatically size the table to available and needed space.

The #maximum_width allows for fluid-width tables (e.g., SimpleTable#width = 0) that do not exceed a certain amount of space. The default value is zero (0), indicating that there is no maximum width. This value will only come into effect if the table will be larger than this value.

If the amount of #width specified for all columns (Column#width) is larger than #width or the #maximum_width, the behaviour is undefined.

SimpleTable#row_gap, SimpleTable#row_gap=

The space added to the bottom of each row between the text and the lines of the cell. Default: 2 units.

Pagination Options

These options tell SimpleTable what to do when the table crosses a page boundary. All measurements here are in PDF userspace units.

SimpleTable#minimum_space, SimpleTable#minimum_space=

This is the minimum amount of space between the bottom of each row and the bottom margin. If the amount of space remaining is less than this, a new page will be started. Default: 100 units.

SimpleTable#protect_rows, SimpleTable#protect_rows=

The number of rows to hold with the heading on the page. If there are less than this number of rows on the page, then the heading and the set of rows will be moved onto a new page. By default, there is one row protected.

SimpleTable#split_rows, SimpleTable#split_rows=

If true, rows may be split across page boundaries. If false (the default), rows will be forced to be on a single page.

SimpleTable#header_gap, SimpleTable#header_gap=

The number of units to leave open at the top of a new page relative to the top margin created during the rendering of a SimpleTable. This is typically used for a repeating text header or other similar items. Default: 0 units.

Drawing the Table

The table is drawn with the #render_on method.

SimpleTable#render_on(pdf)

Draws the table on the provided PDF canvas.

Examples of SimpleTable Classes

What follows are a number of example tables illustrating some of the options available to the SimpleTable class. For all of these, the basic table is the following list of Lord of the Rings characters.

```
table = PDF::SimpleTable.new
table.data = [
  { "row" => 1, "name" => "Gandalf",  "race" => "Human" },
  { "row" => 2, "name" => "Bilbo",    "race" => "Hobbit",
     "url" => "http://www.glyphweb.com/arda/b/bilbobaggins.html", },
  { "row" => 3, "name" => 'Frodo',    "race" => "Hobbit" },
  { "row" => 4, "name" => 'Saruman',  "race" => "Human",
     "url" => "http://www.lord-of-the-rings.org/books/saruman.html",
},
  { "row" => 5, "name" => 'Sauron',   "race" => "???" },
]
```

Basic Table

This table shows the “row,” “name,” and “race” columns with default settings.

```
table.column_order = [ "row", "name", "race" ]
table.render_on(pdf)
```

row	name	race
1	Gandalf	Human
2	Bilbo	Hobbit
3	Frodo	Hobbit
4	Saruman	Human
5	Sauron	???

This one changes the columns displayed and the order in which they display.

```
table.column_order = [ "race", "name" ]
table.render_on(pdf)
```

race	name
Human	Gandalf
Hobbit	Bilbo
Hobbit	Frodo

race	name
Human	Saruman
???	Sauron

Table with Custom Column Headings

This table creates custom column headings for display. For completeness' sake, a title has been added to the table.

```
table.column_order = [ "race", "name" ]
table.columns["race"] = PDF::SimpleTable::Column.new("race") { |col|
  col.heading = "Species"
}
table.columns["name"] = PDF::SimpleTable::Column.new("name") { |col|
  col.heading = "<i>Name</i>"
}
table.title = "Characters from <i>The Lord of the Rings</i>"
table.render_on(pdf)
```

Characters from The Lord of the Rings

Species	Name
Human	Gandalf
Hobbit	Bilbo
Hobbit	Frodo
Human	Saruman
???	Sauron

No Headings, Shadings, or Lines

This table isn't quite the same as the one above. Close, though. Sort of.

```
table.column_order = [ "race", "name" ]
table.columns["race"] = PDF::SimpleTable::Column.new("race") { |col|
  col.heading = "Species"
}
table.columns["name"] = PDF::SimpleTable::Column.new("name") { |col|
  col.heading = "<i>Name</i>"
}
table.title = "Characters from <i>The Lord of the Rings</i>"
table.shade_rows    = :none
table.show_headings = false
table.show_lines    = :none
table.render_on(pdf)
```

Characters from The Lord of the Rings

Human	Gandalf
Hobbit	Bilbo
Hobbit	Frodo
Human	Saruman
???	Sauron

Controlled Width and Positioning

A version of the table with an explicitly too-small width, forcing content to wrap. This table is positioned at the right margin and oriented to the left of the right margin.

```
table.column_order = [ "race", "name" ]
table.columns["race"] = PDF::SimpleTable::Column.new("race") { |col|
  col.heading = "Species"
}
table.columns["name"] = PDF::SimpleTable::Column.new("name") { |col|
  col.heading = "<i>Name</i>"
}
table.position      = :right
table.orientation   = :left
table.width         = 100
table.render_on(pdf)
```

Species	Name
Human	Gandalf
Hobbit	Bilbo
Hobbit	Frodo
Human	Saruman
???	Sauron

This table has a long heading name with a newline embedded. The “row” column is right justified and the “name” column is 100 units, with a size limit on the table of 300 units. The table is positioned at 90 and oriented to the right of that position.

```
table.column_order = %w(row name race)
table.columns["row"] = PDF::SimpleTable::Column.new("row") { |col|
  col.heading = "A Very Long Way Of Saying This Column\nHas Row Numbers"
  col.justification = :right
}
table.columns["name"] = PDF::SimpleTable::Column.new("name") { |col|
  col.heading = "Name"
  col.width = 100
}
table.columns["race"] = PDF::SimpleTable::Column.new("race") { |col|
  col.heading = "Race"
}
table.position      = 90
table.orientation   = :right
table.width         = 300
table.render_on(pdf)
```

A Very Long Way Of Saying This Column Has Row Numbers	Name	Race
1	Gandalf	Human
2	Bilbo	Hobbit
3	Frodo	Hobbit
4	Saruman	Human
5	Sauron	???

Expanding Hyperlinked Table

This will expand the table to a width of 400 units, and the “name” column will use the “url”

column to make a hyperlink if there is one.

```
table.column_order = [ "race", "name" ]
table.columns["name"] = PDF::SimpleTable::Column.new("name") { |col|
  col.link_name = "url"
}
table.show_headings = false
table.shade_rows    = false
table.width         = 400
table.render_on(pdf)
```

Human	Gandalf
Hobbit	Bilbo
Hobbit	Frodo
Human	Saruman
???	Sauron

8. PDF Charts

PDF::Writer comes with a collection of chart drawing classes for the placement of charts on PDF documents. As of this version, only a standard deviation chart is included. Other chart types will be added in future releases.

Standard Deviation Chart (PDF::Charts::StdDev)

This chart will plot an average value on a scale as a point; above and below the average value will be placed the standard deviation from the average value as a vertical bar and horizontal crossbar. The chart will be a fixed height displaying as many data points as it can horizontally to the maximum width. When it cannot display any further data points, it will split the chart and display the remaining data points as a second instance of the chart.

This chart was designed by Cris Ewing of the University of Washington for the R&OS PDF class. It has been adapted to PDF::Writer with some improvements in configurability and capability.

PDF::Charts::StdDev.new

This will create the standard deviation chart with default values. If a block is provided, the created chart will be yielded.

```
chart = PDF::Charts::StdDev.new
```

#data, #data=

This is an array of StdDev::DataPoint objects. This may not be empty when rendering the chart on a PDF document.

```
  # This is sample data for the example chart shown in this chapter.
chart.data << PDF::Charts::StdDev::DataPoint.new(1,      4.0000, 0.5774)
           << PDF::Charts::StdDev::DataPoint.new(2,      4.8333, 0.3727)
           << PDF::Charts::StdDev::DataPoint.new(3,      3.8333, 0.3727)
           << PDF::Charts::StdDev::DataPoint.new(4,      4.0000, 0.5774)
           << PDF::Charts::StdDev::DataPoint.new(5,      4.3333, 0.7454)
           << PDF::Charts::StdDev::DataPoint.new(6,      3.8000, 0.4000)
           << PDF::Charts::StdDev::DataPoint.new(7,      4.1667, 0.8975)
           << PDF::Charts::StdDev::DataPoint.new(8,      4.0000, 0.8165)
           << PDF::Charts::StdDev::DataPoint.new("Tot.", 4.1277, 0.7031)
```

#scale, #scale=

The scale of the chart, a StdDev::Scale object. This must be set to render the chart on a PDF document.

```
    # This is the default chart.
PDF::Charts::StdDev::Scale.new do |scale|
  scale.range               = 0..6
  scale.step                = 1
  scale.style               = PDF::Writer::StrokeStyle.new(0.25)
  scale.show_labels         = false
  PDF::Charts::StdDev::Label.new do |label|
    label.text_size         = 8
    label.text_color        = Color::RGB::Black
    label.pad               = 2
    label.decimal_precision = 1
    scale.label             = label
```

```
  end
  chart.scale              = scale
end
```

#leading_gap, #leading_gap=

The minimum number of userspace units between the chart and the bottom of the page.

```
  # This is the default leading gap.
chart.leading_gap = 10
```

#show_labels, #show_labels=

This should be set to true if labels are to be displayed.

```
  # This is the default.
chart.show_labels = true
```

#label, #label=

The label style for horizontal groupings of data; this must be a StdDev::Label object.

```
  # This is the default chart label style.
Label.new do |label|
  label.height            = 25
  label.background_color  = Color::RGB::Black
  label.text_color        = Color::RGB::White
  label.text_size         = 12
  chart.label = label
end
```

#inner_borders, #inner_borders=
#outer_borders, #outer_borders=

The inner or outer border style. If nil, the unset borders are not drawn. If set, they must be PDF::Charts::StdDev::Marker objects.

```
  # These are the defaults.
PDF::Charts::StdDev::Marker.new do |marker|
  marker.style        = PDF::Writer::StrokeStyle.new(1.5)
  marker.color        = Color::RGB::Black
  chart.outer_borders = marker
end
chart.inner_borders = nil
```

#dot, #dot=
#bar, #bar=
#upper_crossbar, #upper_crossbar=
#lower_crossbar, #bar=

These items will not be drawn if they are nil. If set, they must be PDF::Charts::StdDev::Marker objects.

The #dot displays the level at which average is reached with a filled circle. The #bar is drawn vertically through the average #dot marker (if drawn) from the upper to lower standard deviation. The #upper_crossbar and the #lower_crossbar will be drawn across the top and bottom of the standard deviation #bar. If #dot is nil, the line will be drawn twice as wide as it is thick.

```
  # These are the defaults.
```

```
PDF::Charts::StdDev::Marker.new do |marker|
  marker.style  = PDF::Writer::StrokeStyle.new(5)
  marker.color  = Color::RGB::Black
  chart.dot     = marker
end
PDF::Charts::StdDev::Marker.new do |marker|
  marker.style  = PDF::Writer::StrokeStyle.new(0.5)
  marker.color  = Color::RGB::Black
  chart.bar     = marker
end
PDF::Charts::StdDev::Marker.new do |marker|
  marker.style          = PDF::Writer::StrokeStyle.new(1)
  marker.color          = Color::RGB::Black
  chart.upper_crossbar  = marker
end
PDF::Charts::StdDev::Marker.new do |marker|
  marker.style          = PDF::Writer::StrokeStyle.new(1)
  marker.color          = Color::RGB::Black
  chart.lower_crossbar  = marker
end
```

#height, #height=
#datapoint_width, #datapoint_width=
#maximum_width, #maximum_width=

The chart will be fit into the specified #height; each data point will be drawn with a width of #datapoint_width as long as the chart is less than specified in #maximum_width. After the chart is that large, it will be split and the rest will be displayed as a separate chart.

```
  # These are the defaults.
chart.height          = 200
chart.maximum_width   = 500
chart.datapoint_width = 35
```

#render_on(pdf)
Draw the standard deviation chart on the supplied PDF document.

PDF::Charts::StdDev::DataPoint
A data point for drawing on the chart.

#label, #label=
The label that will be displayed for each datapoint in the X axis of the chart.

#average, #average=
The average (mean) value for each datapoint.

#stddev, #stddev=
The standard deviation for each datapoint.

PDF::Charts::StdDev::Label
A label format for displaying the scale (Y-axis) of data or the data point identifiers (X-axis).

PDF::Charts::StdDev::Label.new

Creates a new Label object. If a block is provided, the created Label is yielded.

#height, #height=
The height of the label, in PDF user units. Ignored for Scale labels.

#background_color, #background_color=
The background color of the label. Ignored for Scale labels.

#text_color, #text_color=
The text color of the label.

#text_size, #text_size=
The text size, in points, of the label.

#pad, #pad=
The padding of the label. Only used for Scale labels.

#decimal_precision, #decimal_precision=
The decimal precision of the label. Only used for Scale labels.

PDF::Charts::StdDev::Scale
The scale (Y-axis) of the dataset.

PDF::Charts::StdDev::Scale.new(args = {})
Creates a new Scale object. Yields the created Scale object when a block is provided. Errors will be raised if the created Scale either does not have a step or range value. Values may be provided to Scale.new as :range (#range), :step (#step), and :style (#style). Labels are not displayed by default in the Y axis of the chart.

#range, #range=
The range of the Scale. This should be a Range object that can meaningfully be counted by step. Generally, this means a numeric value.

#first, #first=
#last, #last=
The first and last values of the Scale’s range. Modifying these values readjust the range.

#step, #step=
How the #range will be subdivided. Each #step represents a vertical position that demarcates the scale visually.

#style, #style=
The line style (as a PDF::Writer::StrokeStyle object) for the horizontal lines demarcating the vertical steps of the scale. If this is nil, there will be no scale demarcation on the chart proper.

#show_labels, #show_labels=
Set to true if the scale labels should be displayed. One label will be displayed for each step and the boundaries of the #range.

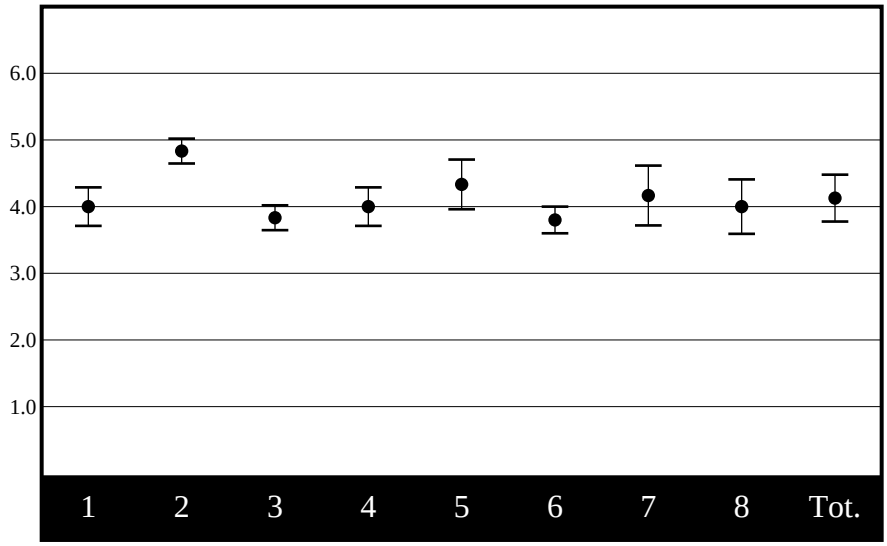
#label, #label=
Defines the label style for the scale labels. If set, this must be a PDF::Charts::StdDev::Label object.

PDF::Charts::StdDev::Marker
This is any line that will be drawn (except scale markers); this is a combination of the line style (which must be a PDF::Writer::StrokeStyle object) and a color. The Marker object will be yielded if a block has been given.

#style, #style=
The StrokeStyle object for the drawn line.

#color, #color=
The colour in which to draw the line.

Example Standard Deviation Chart



9. Quick Reference Sheets and Brochures

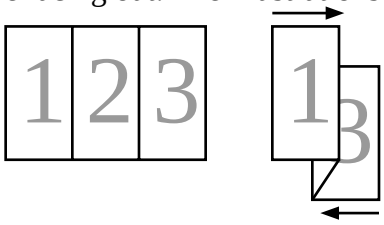
The PDF::Writer distribution includes a class that provides a formatting language to create a quick reference sheet. This is a multi-column page in a landscape layout that generally has three or four columns. This class may also be used for brochures, but brochure creation requires a bit of management to create properly.

Reference sheets and brochures are usually printed doublesided; in the examples below, columns 1–3 are assumed to be on the “front” and 4–6 are assumed to be on the “back.”

Reference Sheets

Refererence sheets are most often three columns, but may be four, and are generally linear in nature. This means that the text flows from the first column to the last column and the reference sheet will be folded such that the first column is the first visible item in the folded page.

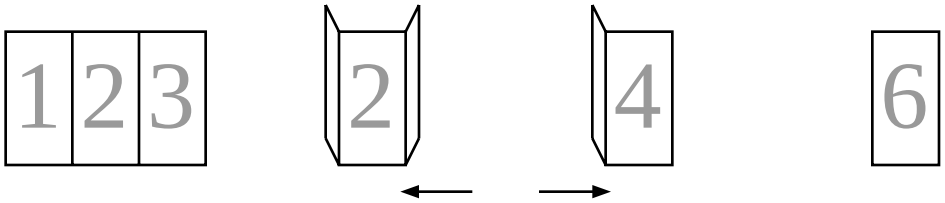
The formatting language provided in PDF::QuickRef is based around this text flow. The title of the quick reference sheet is in column 1. The two pages are intended to be printed on both sides of pieces of paper so that columns 1 and 6 are matched. This will use a Z-fold that places columns 5 and 6 face to face and columns 2 and 3 face to face. In the folded reference sheet, columns 1 and 4 will be facing out. The illustrations below are useful for understanding this.



Brochures

Brochures differ in their design and intent than a reference sheet. A common brochure is also three columns (although parts of a brochure—especially images—may span multiple columns), but the layout differs depending on whether the Z-fold described above is desired or an overlapping fold is desired.

When an overlapping fold is used, the title is typically on column 6 (assuming a left-to-right reading order). A short summary will appear on column 4. Contact information about the maker of the brochure is typically in column 5. Columns 1, 2, and 3 will contain the main body of the brochure. The brochure will be folded so that columns 2 and 3 are face to face. After this, column 1 will face column 4 (exposed by the first fold). In the folded brochure, columns 5 and 6 are facing out. The illustrations below are useful for understanding this.



Using PDF::QuickRef

PDF::QuickRef implements a domain-specific language (DSL) for creating reference sheets or brochures. It is intended to flow from column 1 to column 6 (on a three-column brochure); overlapping fold brochures must be managed explicitly.

The simplest PDF::QuickRef is:

```
qr = PDF::QuickRef.new # 3-column LETTER
qr.title "My QuickRef"
qr.h1 "H1 Text"
qr.lines "Text to put after the header."
qr.save_as "MyQuickRef.pdf"
```

3<PDF::QuickRef.new(paper = "LETTER", columns = 3) Create the quick reference document. The paper parameter is passed unchanged to PDF::Writer.new; the page is always created landscape. Margins are initialized to 18 points. After some additional initialization is performed, the quick reference document is yielded to an optional block for further configuration. All of this is done before the columns are started.

After the columns are started, lines will be drawn between column positions.

QuickRef#pdf

Provides access to the underlying PDF canvas for normal PDF::Writer operations.

QuickRef#title_font, QuickRef#title_font= QuickRef#title_font_encoding, QuickRef#title_font_encoding= QuickRef#title_font_size, QuickRef#title_font_size=

The name, encoding, and size of the font that will be used when drawing title text with #title. The default font is Times-Roman 14 point with the standard ‘WinAnsiEncoding’ encoding.

QuickRef#heading_font, QuickRef#heading_font= QuickRef#heading_font_encoding, QuickRef#heading_font_encoding= QuickRef#h1_font_size, QuickRef#h1_font_size= QuickRef#h2_font_size, QuickRef#h2_font_size= QuickRef#h3_font_size, QuickRef#h3_font_size= QuickRef#h4_font_size, QuickRef#h4_font_size=

The name, encoding, and size of the font that will be used when drawing heading text with #h1, #h2, #h3, or #h4. The default font is Times-Roman with ‘WinAnsiEncoding’ as the encoding. #h1 defaults to 11 points; #h2 to 9 points, #h3 to 8 points, and #h4 to 7 points.

QuickRef#body_font, QuickRef#body_font=
QuickRef#code_font, QuickRef#code_font=
QuickRef#body_font_encoding, QuickRef#body_font_encoding=
QuickRef#code_font_encoding, QuickRef#code_font_encoding=
QuickRef#body_font_size, QuickRef#body_font_size=

Text drawn with #body, #lines, and #pairs will be drawn with the #body_font using #body_font_encoding at #body_font_size. Text drawn with #pre, #codelines, and #codepairs will be drawn with the #code_font using #code_font_encoding at #body_font_size. #body_font defaults to Times-Roman; #code_font to Courier. Both use the 'WinAnsiEncoding' as the default. The default #body_font_size is 7 points.

QuickRef#pairs(text)
QuickRef#codepairs(text)

Creates a two-column shaded table using #body_font (#pairs) or #code_font (#codepairs). Each line of the text is a separate row. The two columns are separated by tab characters.

QuickRef#lines(text)
QuickRef#codelines(text)

Creates a one-column shaded table using #body_font (#pairs) or #code_font (#codepairs). Each line of the text is a separate row.

QuickRef#title(text)

Writes the text with the #title_font and #title_font_size centered in the column. After the title has been written, an #hline will be drawn under the title. The font is set to #body_font after the title is drawn.

QuickRef#h1(text)

Writes the +text+ with the #heading_font and #h1_font_size left justified in the column. The font is set to #body_font after the heading is drawn.

QuickRef#h2(text)

Writes the +text+ with the #heading_font and #h2_font_size left justified in the column. The font is set to #body_font after the heading is drawn.

QuickRef#h3(text)

Writes the +text+ with the #heading_font and #h3_font_size left justified in the column. The font is set to #body_font after the heading is drawn.

QuickRef#h4(text)

Writes the +text+ with the #heading_font and #h4_font_size left justified in the column. The font is set to #body_font after the heading is drawn.

QuickRef#body(text)

Writes body text. Paragraphs will be reflowed for optimal placement of text. Text separated by two line separators (e.g., \n\n, although the separator is platform dependent). The text will be placed with full justification.

QuickRef#pre(text)

Writes code text. Newlines and spaces will be preserved.

QuickRef#hline(style = PDF::Writer::StrokeStyle::DEFAULT, color = Color::RGB::Black)

Draws a horizontal line with the specified style and colour across the width of the column.

QuickRef#save_as(filename)

Writes the Quick Reference to disk.

QuickRef#render, QuickRef#to_s

Generates the PDF document as a string.

3<QuickRef.make(paper, columns, &block) Creates a QuickRef document and then calls #instance_eval on the document with the supplied block. This allows for a more natural use of the QuickRef class as a DSL for creating these documents.

```
PDF::QuickRef.make do # 3-column LETTER
  title "My QuickRef"
  h1 "H1 Text"
  lines "Text to put after the header."
  save_as "MyQuickRef.pdf"
end
```

10. PDF::TechBook

The TechBook class is a markup language interpreter. This will read a file containing the "TechBook" markukp, described below, and create a PDF document from it. This is intended as a complete document language, but it does have a number of limitations.

The TechBook markup language and class are used to format the PDF::Writer manual, represented in the distrubtion by the file "manual.pwd".

The TechBook markup language is primarily stream-oriented with awareness of lines. That is to say that the document will be read and generated from beginning to end in the order of the markup stream.

TechBook Markup

TechBook markup is relatively simple. The simplest markup is no markup at all (flowed paragraphs). This means that two lines separated by a single line separator will be treaed as part of the same paragraph and formatted appropriately by PDF::Writer. Paragaphs are terminated by empty lines, valid line markup directives, or valid headings.

Certain XML entitites will need to be escaped as they would in normal XML usage, that is, "<" must be written as "<"; ">" must be written as ">"; and "&" must be written as "&".

Comments, headings, and directives are line-oriented where the first mandatory character is in the first column of the document and take up the whole line. Styling and callback tags may appear anywhere in the text.

Comments

Comments begin with the hash-mark ('#') at the beginning of the line. Comment lines are ignored.

Styling and Callback Tags

Within normal, preserved, or code text, or in headings, HTML-like markup may be used for bold ("<b>") and italic ("<i>") text. TechBook supports standard PDF::Writer callback tags (<c:alink>, <c:ilink>, <C:bullet/>, and <C:disc/>) and adds two new ones (<r:xref/>, <C:tocdots/>). See " " for more information.

Directives

Directives begin with a period ('.') and are followed by a letter ('a'..'z') and then any combination of word characters ('a'..'z', '0'..'9', and '_'). Directives are case-insensitive. A directive may have arguments; if there are arguments, they must follow the directive name after whitespace. After the arguments for a directive, if any, all other text is ignored and may be considered a comment.

.newpage [force]

The .newpage directive starts a new page. If multicolumn mode is on, a new column will be started if the current column is not the last column. If the optional argument force follows the .newpage

directive, a new page will be started even if multicolumn mode is on.

```
.newpage
.newpage force
```

.pre, .endpre

The .pre and .endpre directives enclose a block of text with preserved newlines. This is similar to normal text, but the lines in the .pre block are not flowed together. This is useful for poetic forms or other text that must end when each line ends. .pre blocks may not be nested in any other formatting block. When an .endpre directive is encountered, the text format will be returned to normal (flowed text) mode.

```
.pre
The Way that can be told of is not the eternal Way;
The name that can be named is not the eternal name.
The Nameless is the origin of Heaven and Earth;
The Named is the mother of all things.
Therefore let there always be non-being,
  so we may see their subtlety,
And let there always be being,
  so we may see their outcome.
The two are the same,
But after they are produced,
  they have different names.
.endpre
```

This will look like the following:

The Way that can be told of is not the eternal Way;
The name that can be named is not the eternal name.
The Nameless is the origin of Heaven and Earth;
The Named is the mother of all things.
Therefore let there always be non-being,
  so we may see their subtlety,
And let there always be being,
  so we may see their outcome.
The two are the same,
But after they are produced,
  they have different names.

.code, .endcode

The .code and .endcode directives enclose a block of text with preserved newlines. In addition, the font is changed from the normal #techbook_textfont to #techbook_codefont. The #techbook_codefont is normally a fixed pitched font and defaults to Courier. At the end of the code block, the text state is restored to its prior state, which will either be .pre or normal.

```
.code
require 'pdf/writer'
PDF::Writer.prepress # US Letter, portrait, 1.3, prepress
.endcode
```

.blist, .endblist

These directives enclose a bulleted list block. Lists may be nested within other text states. If lists are nested, each list will be appropriately indented. Each line in the list block will be treated as a single

list item with a bullet inserted in front using either the <C:bullet/> or <C:disc/> callbacks. Nested lists are successively indented. .blist directives accept one optional argument, the name of the type of bullet callback desired (e.g., 'bullet' for <C:bullet/> and 'disc' for <C:disc/>).

```
.blist
Item 1
.blist disc
Item 1.1
.endblist
.endblist
```

.eval, .endeval

With these directives, the block enclosed will collected and passed to Ruby's Kernel#eval. .eval blocks may be present within normal text, .pre, .code, and .blist blocks. No other block may be embedded within an .eval block.

```
.eval
puts "Hello"
.endeval
```

.columns

Multi-column output is controlled with this directive, which accepts one or two parameters. The first parameter is mandatory and is either the number of columns (2 or more) or the word 'off' (turning off multi-column output). When starting multi-column output, a second parameter with the gutter size may be specified.

```
.columns 3
Column 1
.newpage
Column 2
.newpage
Column 3
.columns off
```

.toc [TITLE]

This directive is used to tell TechBook to generate a table of contents after the first page (assumed to be a title page). If this is not present, then a table of contents will not be generated. The title of the table of contents can be provided after this directive.

.author, .title, .subject, .keywords

Sets values in the PDF information object. The arguments—to the end of the line—are used to populate the values.

.done

Stops the processing of the document at this point.

Headings

Headings begin with a number followed by the rest of the heading format. This format is "#<heading-text>" or "#<heading-text>xref_name". TechBook supports five levels of headings. Headings may include markup, but should not exceed a single line in size; those headings which have boxes as part of their layout are not currently configured to work with multiple lines of heading output. If an xref_name is specified, then the <r:xref/> tag can use this name

to find the target for the heading. If xref_name is not specified, then the “name” associated with the heading is the index of the order of insertion. The xref_name is case sensitive.

```
1<Chapter>xChapter
2<Section>Section23
3<Subsection>
4<Subsection>
5<Subsection>
```

Heading Level 1

First level headings are generally chapters. As such, the standard implementation of the heading level 1 method (#__heading1), will be rendered as “chapter#. heading-text” in centered white on a black background, at 26 point (H1_STYLE). First level headings are added to the table of contents.

Heading Level 2

Second level headings are major sections in chapters. The headings are rendered by default as black on 80% grey, left-justified at 18 point (H2_STYLE). The text is unchanged (#__heading2). Second level headings are added to the table of contents.

Heading Level 3, 4, and 5

The next three heading levels are used for varying sections within second level chapter sections. They are rendered by default in black on the background (there is no bar) at 18, 14, and 12 points, respectively (H3_STYLE, H4_STYLE, and H5_STYLE). Third level headings are bold-faced (#__heading3); fourth level headings are italicised (#__heading4), and fifth level headings are underlined (#__heading5).

Configuring TechBook

TechBook is reasonably configurable. Formatting options may be changed with methods in the techbook object. Text transformations and directives must be done either as a subclass of TechBook or by reopening the TechBook class.

Formatting Options

These options are configurable as a client and can be configured within the document source with an .eval directive.

#techbook_codefont
#techbook_textfont
#techbook_encoding
#techbook_fontsize

These values configure the fonts and encoding for TechBook documents.

Code Extensions

techbook Command

The TechBook class is also available with a command-line program, techbook (bin/techbook). The command-line interface is:

```
Usage: techbook [options] [INPUT FILE]
```

```
  INPUT FILE, if not specified, will be 'manual.pwd', either in the
  current directory or relative to this file.
    -f, --force-regen               Forces the regeneration of the
document,
                                    ignoring the cached document version.
    -n, --no-cache                  Disables generated document caching.
    -z, --compress                  Compresses the resulting PDF.
        --help                      Shows this text.
```

--no-cache

By default, the techbook command will generate the PDF document and a cached version of the document. If the input file is “test.pwd”, then both “test.pdf” and “test._mc” will be generated. This can be disabled by using the --no-cache option.

--force-regen

If the input file is older than the generated cache document (if one exists), then techbook command will regenerate the document from the input file. This option forces this regeneration.

--compress

The document will not be compressed by default. This option will compress the document. This can only be applied if the document is being generated from the original document, not the cached document.

11. PDF::Writer Dependencies

PDF::Writer uses color-tools and Transaction::Simple. These packages must be installed before PDF::Writer can be used. Because these packages are heavily used in PDF::Writer, the documentation about these packages has been included in this manual.

Transaction::Simple

This package provides "active object transaction support." In a nutshell, this means that an object can be modified and reverted to its original state if there is a problem with the modification. Transaction::Simple transactions will work with most objects, but requires that the supported objects can be Marshaled (which explicitly excludes singleton objects).

These transactions are not backed by a data store and are not in support of a data store; they are "live" and in memory. The typical use of Transaction::Simple objects is to allow changes to be tested against an object prior to making the changes permanent.

Multiple levels of transactions are supported. There is no algorithmic limit on the number of transaction levels (it is limited only by memory). Transactions may be positional (known only by the order in which the transaction was opened) or "named" so that a transaction may be manipulated from an arbitrary depth while simultaneously affecting all transactions opened after the named transaction. Transactions names may be any object except nil.

Anonymous Transactions

```
include 'transaction/simple'
v = "Hello, you."                # -> "Hello, you."
v.extend(Transaction::Simple)    # -> "Hello, you."
v.start_transaction              # -> ... (a Marshal string)
v.transaction_open?              # -> true
v.gsub!(/you/, "world")          # -> "Hello, world."
v.rewind_transaction             # -> "Hello, you."
v.transaction_open?              # -> true
v.gsub!(/you/, "HAL")            # -> "Hello, HAL."
v.abort_transaction              # -> "Hello, you."
v.transaction_open?              # -> false
v.start_transaction              # -> ... (a Marshal string)
v.start_transaction              # -> ... (a Marshal string)
v.transaction_open?              # -> true
v.gsub!(/you/, "HAL")            # -> "Hello, HAL."
v.commit_transaction             # -> "Hello, HAL."
v.transaction_open?              # -> true
v.abort_transaction              # -> "Hello, you."
v.transaction_open?              # -> false
```

Named Transactions

```
v = "Hello, you."                # -> "Hello, you."
v.extend(Transaction::Simple)    # -> "Hello, you."
v.start_transaction(:first)      # -> ... (a Marshal string)
v.transaction_open?              # -> true
v.transaction_open?(:first)      # -> true
v.transaction_open?(:second)     # -> false
v.gsub!(/you/, "world")          # -> "Hello, world."
v.start_transaction(:second)     # -> ... (a Marshal string)
v.gsub!(/world/, "HAL")          # -> "Hello, HAL."
v.rewind_transaction(:first)     # -> "Hello, you."
```

```
v.transaction_open?             # -> true
v.transaction_open?(:first)     # -> true
v.transaction_open?(:second)    # -> false
v.gsub!(/you/, "world")         # -> "Hello, world."
v.start_transaction(:second)    # -> ... (a Marshal string)
v.gsub!(/world/, "HAL")         # -> "Hello, HAL."
v.transaction_name              # -> :second
v.abort_transaction(:first)     # -> "Hello, you."
v.transaction_open?             # -> false
v.start_transaction(:first)     # -> ... (a Marshal string)
v.gsub!(/you/, "world")         # -> "Hello, world."
v.start_transaction(:second)    # -> ... (a Marshal string)
v.gsub!(/world/, "HAL")         # -> "Hello, HAL."
v.commit_transaction(:first)    # -> "Hello, HAL."
v.transaction_open?             # -> false
```

Block Transactions

```
v = "Hello, you."               # -> "Hello, you."
Transaction::Simple.start(v) do |tv|
    # v has been extended with Transaction::Simple and an unnamed
    # transaction has been started.
  tv.transaction_open?          # -> true
  tv.gsub!(/you/, "world")      # -> "Hello, world."
  tv.rewind_transaction         # -> "Hello, you."
  tv.transaction_open?          # -> true
  tv.gsub!(/you/, "HAL")        # -> "Hello, HAL."
    # The following breaks out of the transaction block after
    # aborting the transaction.
  tv.abort_transaction          # -> "Hello, you."
end
  # v still has Transaction::Simple applied from here on out.
v.transaction_open?             # -> false
Transaction::Simple.start(v) do |tv|
  tv.start_transaction          # -> ... (a Marshal string)
  tv.transaction_open?          # -> true
  tv.gsub!(/you/, "HAL")        # -> "Hello, HAL."
    # If #commit_transaction were called without having started a
    # second transaction, then it would break out of the transaction
    # block after committing the transaction.
  tv.commit_transaction         # -> "Hello, HAL."
  tv.transaction_open?          # -> true
  tv.abort_transaction          # -> "Hello, you."
end
v.transaction_open?             # -> false
```

Grouped Transactions

```
require 'transaction/simple/group'
x = "Hello, you."
y = "And you, too."
g = Transaction::Simple::Group.new(x, y)
g.start_transaction(:first)     # -> [ x, y ]
g.transaction_open?(:first)     # -> true
x.transaction_open?(:first)     # -> true
y.transaction_open?(:first)     # -> true
x.gsub!(/you/, "world")         # -> "Hello, world."
y.gsub!(/you/, "me")            # -> "And me, too."
g.start_transaction(:second)    # -> [ x, y ]
x.gsub!(/world/, "HAL")         # -> "Hello, HAL."
y.gsub!(/me/, "Dave")           # -> "And Dave, too."
g.rewind_transaction(:second)   # -> [ x, y ]
x                               # -> "Hello, world."
y                               # -> "And me, too."
```

```
x.gsub!(/world/, "HAL")         # -> "Hello, HAL."
y.gsub!(/me/, "Dave")           # -> "And Dave, too."
g.commit_transaction(:second)   # -> [ x, y ]
x                               # -> "Hello, HAL."
y                               # -> "And Dave, too."
g.abort_transaction(:first)     # -> [ x, y ]
x                               = -> "Hello, you."
y                               = -> "And you, too."
```

Methods

These are the methods that objects extended with Transaction::Simple will know.

#transaction_open?(name = nil)

Returns true if a transaction is currently open if the name parameter is nil. If a name is requested, then returns true if a transaction with that name is currently open.

#transaction_name

Returns the name of the current transaction. Transactions not explicitly named will return nil.

#start_transaction(name = nil)

Starts a transaction. If a name is provided, the transaction will be known by that name. If not, this will be an anonymous transaction.

#rewind_transaction(name = nil)

Rewinds the current or named transaction, which means that the state of the object is restored to the state it had when the transaction was started, but the transaction remains open. With a named transaction, intervening transactions will be aborted and the named transaction is rewound. Otherwise, only the current transaction is rewound.

#abort_transaction(name = nil)

Aborts the current or named transaction. This resets the object state to what it was when the transaction was started and closes the transaction. With a named transaction, intervening transactions and the named transaction will be aborted. Otherwise, only the current transaction is aborted.

If the current or named transaction has been started by a block (Transaction::Simple.start), then the execution of the block will be halted with break self.

#commit_transaction(name = nil)

Commits the current or named transaction. This closes the transaction, saving the object in its current state. With a named transaction, all intervening transactions are closed as well.

#transaction_exclusions

This value allows specific variables to be excluded from transaction support for the object under transaction support. This must be modified after extending the object but before starting the first transaction on the oject.

Transaction::Simple.start_named(name, *vars, &block)

Transaction::Simple.start(*vars, &block)

Starts a named or anonymous transaction against one or more variables that will be run in a block.

When the block completes, the transactions will automatically be committed. If the transaction that started the block is aborted, the block itself will be aborted. If the transaction that started the block is committed, the block itself will be stopped.

Transaction Groups

A transaction group is an object wrapper that manages a group of objects as if they were a single object for the purpose of transaction management. All transactions for this group of objects should be performed against the transaction group object, not against individual objects in the group. Transaction group objects support the standard Transaction::Simple methods but apply them to all objects in the transaction group.

Transaction::Simple::Group.new(*objects)

Creates a transaction group for the provided objects. If a block is provided, the transaction group object is yielded to the block; when the block is finished, the transaction group object will be cleared with #clear.

#objects

Returns the objects that are covered by this transaction group.

#clear

Clears the object group. Removes references to the objects so that they can be garbage collected.

Thread Safety

Transaction::Simple is not inherently threadsafe; a threadsafe version has been provided as Transaction::Simple::ThreadSafe and Transaction::Simple::ThreadSafe::Group, respectively.

Limitations

While Transaction::Simple is very useful, it has some severe limitations that must be understood. Transaction::Simple:

- …uses Marshal. Thus, any object which cannot be Marshaled cannot use Transaction::Simple. In my experience, this affects singleton objects more often than any other object. It may be that Ruby 2.0 will solve this problem.
- …does not manage resources. Resources external to the object and its instance variables are not managed at all. However, all instance variables and objects “belonging” to those instance variables are managed. If there are object reference counts to be handled, Transaction::Simple will probably cause problems.
- …is not inherently thread-safe. In the ACID (“atomic, consistent, isolated, durable”) test, Transaction::Simple provides CD, but it is up to the user of Transaction::Simple to provide isolation and atomicity. Transactions should be considered “critical sections” in multi-threaded applications. If thread safety and atomicity is absolutely required, use Transaction::Simple::ThreadSafe, which uses a Mutex object to synchronize the accesses on the object during the transaction operations.
- …does not necessarily maintain Object#__id__ values on rewind or abort. This may change for future versions that will be Ruby 1.8 or better only. Certain objects that support #replace will maintain Object#__id__.
- …can be a memory hog if you use many levels of transactions on many objects.

color-tools

The color-tools package was created to abstract out the colour needs for PDF::Writer and make them available for other uses as well (such as web pages). There are several main components to color-tools in version 1.3.0:

- Named colours: more than 150 named RGB colours so that the exact colour representation does not need to be remembered.
- Color::RGB: a colour class that encapsulates operations on RGB (red, green, blue) colours.
- Color::CMYK: a colour class that encapsulates operations on CMYK (cyan, magenta, yellow, black) colours, mostly used for printing.
- Color::GrayScale: a colour class that encapsulates operations on greyscale colours, mostly used for PDF operations.
- Color::YIQ: a colour class that encapsulates the NTSC/YIQ video colour space.
- Color::HSL: a colour class that encapsulates the HSL (hue, saturation, luminance) colour space.
- Color::Palette::MonoContrast: an RGB colour palette generator.
- Color::Palette::Gimp: a class to read Gimp (GNU Image Manipulation Program) colour palettes and make them available for access.
- Color::CSS: An interface for interpreting named CSS3 colours.

Named Colors

There are more than 150 named colours available in the Color::RGB and Color::RGB::Metallic namespaces. The demo program demo/colornames.rb shows these colours. The colours in Color::RGB used to be part of the Color namespace. If a colour like Color::Black is used, there will be a single warning printed. Future versions of color-tools will see a warning printed on every use (1.4.x) and ultimately the constants will be completely removed (1.5.x or 2.x).

Color::RGB

This class encapsulates RGB colours and offers some operations on them. Other colours will be coerced into RGB for comparison.

Color::RGB.new(r = 0, g = 0, b = 0)

This will create an RGB colour object from the octet (byte) range of 0 .. 255.

```
blue  = Color::RGB.new(0, 0, 255)
white = Color::RGB.new(0xff, 0xff, 0xff)
```

Color::RGB.from_percentage(r = 0, g = 0, b = 0)

Creates an RGB colour object from percentages (0 .. 100).

```
  # Makes an RGB colour that is 20% red, 30% green, and 75% blue.
mix = Color::RGB.from_percentage(20, 30, 75)
```

Color::RGB.from_fraction(r = 0.0, g = 0.0, b = 0.0)

Creates an RGB colour object from fractional values (0 .. 1).

```
  # Makes an RGB colour that is 20% red, 30% green, and 75% blue.
mix = Color::RGB.from_percentage(0.20, 0.30, 0.75)
```

Color::RGB.from_html(html_colour)

Creates an RGB colour object from HTML and CSS format colour strings. These strings may optionally include the hash mark (‘#’) and semicolon (‘;’, for CSS) and may either be a colour triplet or a colour sextet. According to the HTML colour rules, a colour triplet (e.g., “#fed”) will have each of its elements doubled (“#ffeedd”).

```
fed     = Color::RGB.from_html("fed")
```

```
cabbed  = Color::RGB.from_html("#cabbed;")
```

Color::RGB#==(other)

Compares the other colour to this one. The other colour will be converted to RGB before comparison, so the comparison between a RGB colour and a non-RGB colour will be approximate and based on the other colour's default #to_rgb conversion. If there is no #to_rgb conversion, this will raise an exception. This will report that two RGB colours are equivalent if all component values are within 1e-4 (0.0001) of each other.

Color::RGB#pdf_fill

Color::RGB#pdf_stroke

Renders the colour as a string suitable for PDF colour changes. These two methods will be removed from the color-tools base package in version 2.0.

```
cabbed.pdf_fill
blue.pdf_stroke
```

Color::RGB#html

Renders the colour as an HTML and CSS format colour string. A hash mark ('#') is added to the beginning of the colour string.

```
cabbed.html # -> "#cabbed"
fed.html    # -> "#ffeedd"
```

Color::RGB#to_cmyk

Converts the RGB colour to CMYK. This is an approximation only, and colour experts strongly suggest that this is a bad idea. CMYK colours represent percentages of inks on paper, not mixed colour intensities like RGB. The CMYK versions of RGB colours will usually be darker than the original RGB.

```
cabbed.to_cmyk
```

The method used is as follows:

1. RGB-<CMY

Convert the R, G, and B components to C, M, and Y components.

```
c = 1.0 – r
m = 1.0 – g
y = 1.0 – b
```

2. Minimum Black Calculation

Compute the minimum amount of black (K) required to smooth the colour in inks.

```
k = min(c, m, y)
```

3. Undercolour Removal

Perform undercolour removal on the C, M, and Y components of the colours because less of each colour is needed for each bit of black. Also, regenerate the black (K) based on the undercolour removal so that the colour is more accurately represented in ink.

```
c = min(1.0, max(0.0, c – UCR(k)))
m = min(1.0, max(0.0, m – UCR(k)))
y = min(1.0, max(0.0, y – UCR(k)))
k = min(1.0, max(0.0, BG(k)))
```

The undercolour removal function and the black generation functions return a value based on the brightness of the RGB colour.

Color::RGB#to_rgb(ignored = nil)
Returns the existing colour object.

Color::RGB#to_hsl
Converts the colour to the HSL colour space.

Color::RGB#lighten_by(percent)
Mix the RGB hue with White so that the RGB hue is the specified percentage of the resulting colour. Strictly speaking, this isn't a lighten_by operation.

Color::RGB#darken_by(percent)
Mix the RGB hue with Black so that the RGB hue is the specified percentage of the resulting colour. Strictly speaking, this isn't a darken_by operation.

Color::RGB#mix_with(mask, opacity)
Mixes the mask colour (which must be an RGB object) with the current colour at the stated opacity percentage. This is an imperfect mix; 100% opacity will result in the mask colour.

Color::RGB#to_yiq
Converts the RGB to the YIQ (NTSC) television colour encoding. Because YIQ does not yet exist as a colour object, this is returned as a hash object with keys of :y, :i, and :q.

Color::RGB#brightness
Returns the perceived brightness value for a colour on a scale of 0 .. 1. This is based on the Y (luminosity) value of YIQ colour encoding.

Color::RGB#to_grayscale
Color::RGB#to_greyscale
Converts the color to a GrayScale colour based on the HSL luminance.

Color::RGB#adjust_brightness(percent)
Returns a new colour with the brightness adjusted by the specified percentage. Negative percentages will darken the colour; positive percentages will brighten the colour.

```
Color::RGB::DarkBlue.adjust_brightness(10)
Color::RGB::DarkBlue.adjust_brightness(-10)
```

Color::RGB#adjust_saturation(percent)
Returns a new colour with the saturation adjusted by the specified percentage. Negative percentages will reduce the saturation; positive percentages will increase the saturation.

```
Color::RGB::DarkBlue.adjust_saturation(10)
Color::RGB::DarkBlue.adjust_saturation(-10)
```

Color::RGB#adjust_hue(percent)
Returns a new colour with the hue adjusted by the specified percentage. Negative percentages will reduce the hue; positive percentages will increase the hue.

```
Color::RGB::DarkBlue.adjust_hue(10)
Color::RGB::DarkBlue.adjust_hue(-10)
```

Color::RGB#r, Color::RGB#r=

Color::RGB#g, Color::RGB#g=
Color::RGB#b, Color::RGB#b=

Read and set the independent components of the RGB colour. The colour values will be in the range (0..1).

Color::CMYK

This class encapsulates CMYK colours and offers some operations on them. Other colours will be coerced into CMYK for comparison.

Color::CMYK.new(c = 0, m = 0, y = 0, k = 0)

Creates a CMYK colour object from percentages.

```
black = Color::CMYK.new(0, 0, 0, 100)
```

Color::CMYK.from_fraction(c = 0, m = 0, y = 0, k = 0)

Creates a CMYK colour object from fractional values (0..1).

```
black = Color::CMYK.new(0, 0, 0, 1)
```

Color::CMYK#==(other)

Compares the other colour to this one. The other colour will be converted to CMYK before comparison, so the comparison between a CMYK colour and a non-CMYK colour will be approximate and based on the other colour's #to_cmyk conversion. If there is no #to_cmyk conversion, this will raise an exception. This will report that two CMYK colours are equivalent if all component values are within 1e-4 (0.0001) of each other.

Color::CMYK#pdf_fill
Color::CMYK#pdf_stroke

Renders the colour as a string suitable for PDF colour changes. These two methods will be removed from the color-tools base package in version 2.0.

```
black.pdf_fill
black.pdf_stroke
```

Color::CMYK#html

Present the colour as an RGB HTML/CSS colour string. Note that this will perform a #to_rgb operation using the default conversion formula.

Color::CMYK#to_rgb(use_adobe_method = false)

Converts the CMYK colour to RGB. Most colour experts strongly suggest that this is not a good idea (some even suggesting that it's a very bad idea). CMYK represents additive percentages of inks on white paper, whereas RGB represents mixed colour intensities on a black screen.

However, the colour conversion can be done, and there are two different methods for the conversion that provide slightly different results. Adobe PDF conversions are done with the first form.

```
  # Adobe PDF Display Formula
r = 1.0 - min(1.0, c + k)
g = 1.0 - min(1.0, m + k)
b = 1.0 - min(1.0, y + k)
  # Other
r = 1.0 - (c * (1.0 - k) + k)
g = 1.0 - (m * (1.0 - k) + k)
b = 1.0 - (y * (1.0 - k) + k)
```

If we have a CMYK colour of [33% 66% 83% 25%], the first method will give an approximate RGB colour of (107, 23, 0) or #6b1700. The second method will give an approximate RGB colour of (128, 65, 33) or #804121. Which is correct? Although the colours may seem to be drastically different in the RGB colour space, they are very similar colours, differing mostly in intensity. The first is a darker, slightly redder brown; the second is a lighter brown.

Because of this subtlety, both methods are now offered for conversion in color-tools 1.2 or later. The Adobe method is not used by default; to enable it, pass true to #to_rgb.

Future versions of color-tools may offer other conversion mechanisms that offer greater colour fidelity.

Color::CMYK#to_grayscale
Color::CMYK#to_greyscale

Converts the CMYK colour to a single greyscale value. There are undoubtedly multiple methods for this conversion, but only a minor variant of the Adobe conversion method will be used:

```
g = 1.0 - min(1.0, 0.299 * c + 0.587 * m + 0.114 * y + k)
```

This treats the CMY values similarly to YIQ (NTSC) values and then adds the level of black. This is a variant of the Adobe version because it uses the more precise YIQ (NTSC) conversion values for Y (intensity) rather than the approximates provided by Adobe (0.3, 0.59, and 0.11).

Color::CMYK#to_cmyk

Returns the current CMYK colour object.

Color::CMYK#to_yiq
Color::CMYK#to_hsl

Converts to RGB and then HSL. Uses only the default RGB conversion method.

Color::CMYK#c, Color::CMYK#c=
Color::CMYK#m, Color::CMYK#m=
Color::CMYK#y, Color::CMYK#y=
Color::CMYK#k, Color::CMYK#k=

Read and set the independent components of the CMYK colour. The colour values will be in the range (0..1).

Color::GrayScale

Also know as Color::GreyScale, this is to provide colours representing continuous shades of grey.

Color::GrayScale.from_fraction(g = 0)

Creates a greyscale colour object from fractional (0..1).

```
Color::GreyScale.from_fraction(0.5)
```

4<Color::GrayScale.new(g = 0) Creates a greyscale colour object from percentages (0..100).

```
Color::GrayScale.new(50)
```

Color::GrayScale#==(other)

Compares the other colour to this one. The other colour will be converted to GreyScale before comparison, so the comparison between a GreyScale colour and a non-GreyScale colour will be approximate and based on the other colour's #to_greyscale conversion. If there is no #to_greyscale conversion, this will raise an exception. This will report that two GreyScale values are equivalent if they are within 1e-4 (0.0001) of each other.

Color::GrayScale#pdf_fill
Color::GrayScale#pdf_stroke
Present the colour as a string suitable for PDF colour changes with DeviceGray. These two methods will be removed from the color-tools base package in version 2.0.

Color::GrayScale#html
Presents the colour as an RGB HTML/CSS colour string.

Color::GrayScale#to_cmyk
Convert the greyscale colour to CMYK.

Color::GrayScale#to_rgb(ignored = true)
Convert the greyscale colour to RGB.

Color::GrayScale#to_grayscale
Color::GrayScale#to_greyscale
Returns the current greyscale colour object.

Color::GrayScale#lighten_by
Lightens the greyscale colour by the stated percent.

Color::GrayScale#darken_by
Darkens the greyscale colour by the stated percent.

Color::GrayScale#to_yiq
Returns the YIQ (NTSC) colour encoding of the greyscale value. This is an approximation, as the values for I and Q are calculated by treating the greyscale value as an RGB value. The Y (intensity or brightness) value is the same as the greyscale value.

Color::GrayScale#to_hsl
Returns the HSL colour encoding of the greyscale value.

Color::GrayScale#brightness
Returns the brightness value for this greyscale value; this is the percentage of grey.

Color::GrayScale#g, Color::GrayScale#g=
Read and set the grayscale value. The colour values will be in the range (0..1).

Color::HSL
An HSL colour object. Internally, the hue (#h), saturation (#s), and luminosity (#l) values are dealt with as fractional values in the range (0..1).

Color::HSL.from_fraction(h = 0.0, s = 0.0, l = 0.0)
Creates an HSL colour object from fractional values 0..1.

Color::HSL.new(h = 0, s = 0, l = 0)
Creates an HSL colour object from the standard values of degrees and percentages (e.g., 145º, 30%, 50%).

Color::HSL#==(other)
Compares the other colour to this one. The other colour will be converted to HSL before comparison, so the comparison between a HSL colour and a non-HSL colour will be approximate and based on the other colour's #to_hsl conversion. If there is no #to_hsl conversion, this will raise an exception. This will report that two HSL values are equivalent if all component values are within 1e-4 (0.0001) of each other.

Color::HSL#html
Present the colour as an HTML/CSS colour string.

Color::HSL#to_rgb(ignored = nil)
Converting to HSL as adapted from Foley and Van-Dam from http://www.bobpowell.net/RGBHSB.htm.

Color::HSL#to_yiq
Color::HSL#to_cmyk
Converts to RGB and then YIQ and CMYK, respectively.

Color::HSL#brightness
Returns the luminosity (#l) of the colour.

Color::HSL#to_greyscale
Color::HSL#to_grayscale
Converts the HSL colour to Color::GrayScale.

Color::HSL#h, Color::HSL#h=
Color::HSL#s, Color::HSL#s=
Color::HSL#l, Color::HSL#l=
Read and set the independent components of the HSL colour. The colour values will be in the range (0..1).

Color::YIQ
A colour object representing YIQ (NTSC) colour encoding.

Color::YIQ.from_fraction(y = 0.0, i = 0.0, q = 0.0)
Creates a YIQ colour object from fractional values (0..1).
```
Color::YIQ.new(0.3, 0.2, 0.1)
```

Color::YIQ.new(y = 0, i = 0, q = 0)
Creates a YIQ colour object from percentages (0..100).
```
Color::YIQ.new(10, 20, 30)
```

Color::YIQ#==(other)
Compares the other colour to this one. The other colour will be converted to YIQ before comparison, so the comparison between a YIQ colour and a non-YIQ colour will be approximate and based on the other colour's #to_yiq conversion. If there is no #to_yiq conversion, this will raise an exception. This will report that two YIQ values are equivalent if all component colours are within 1e-4 (0.0001) of each other.

Color::YIQ#to_yiq
Returns the current YIQ colour.

Color::YIQ#brightness
Returns the brightness (#y) of the current colour.

Color::YIQ#to_grayscale
Color::YIQ#to_greyscale
Converts the current YIQ colour to Color::GrayScale based on the #y version.

Color::YIQ#y, Color::YIQ#y=
Color::YIQ#i, Color::YIQ#i=
Coqor::YIQ#q, Coqor::YIQ#q=
Read and set the independent components of the HSL colour. The colour values will be in the range (0..1).

Color::CSS
This namespace contains some CSS colour names.

Color::CSS[name]
Returns the RGB colour for name or +nil+ if the name is not valid. This is based on the constants from the Color::RGB namespace.

Future versions of this will be extended to recognise other valid CSS colour specifications, including hsl(), hsv(), and rgb().

Color::Palette::MonoContrast
Generates a monochromatic constrasting colour palette for background and foreground. What does this mean?

A single colour is used to generate the base palette, and this colour is lightened five times and darkened five times to provide eleven distinct colours (including the base colours). The foreground is also generated as a monochromatic colour palettte; however, all generated colours are tested to see that they are appropriately contrasting to ensure maximum readability of the foreground against the background.

This was developed based on techniques described by Andy "Malarkey" Clarke, implemented in JavaScript by Steve G. Chipman at SlayerOffice and by Patrick Fitzgerald of BarelyFitz in PHP.

Color::Palette::MonoContrast.new(background, foreground = nil)

Generates the palette based on background and foreground. If foreground is nil, then a foreground is chosen based on the background colour.

Color::Palette::MonoContrast#background
Color::Palette::MonoContrast#foreground
The hash of background and foreground colour values. Each is always eleven values.

- [0] is the starting colour.
- [+1] .. [+5] are lighter colours (85%, 75%, 50%, 25%, and 10%).
- [-1] .. [-5] are darker colours (85%, 75%, 50%, 25%, and 10%).

Color::Palette::MonoContrast#minimum_brightness_diff
Color::Palette::MonoContrast#minimum_brightness_diff=
The minimum brightness difference between the background and the foreground, and must be between (0..1). Setting this value will regenerate the palette based on the base colours. The default value for this is 125 / 255.0. If this value is set to nil, it will be restored to the default.

Color::Palette::MonoContrast#minimum_color_diff
Color::Palette::MonoContrast#minimum_color_diff=
The minimum colour difference between the background and the foreground, and must be between 0..3. Setting this value will regenerate the palette based on the base colours. The default value for this is about 1.96 (500 / 255). If this value is set to nil, it will be restored to the default.

4<Color::Palette::MonoContrast#regenerate(background, foreground = nil) Regenerates the palette based on new background and foreground values. If foreground is nil, then a foreground is chosen based on the background colour.

Color::Palette::MonoContrast#calculate_foreground(background, foreground)
Given a background colour and a foreground colour, modifies the foreground colour so that it will have enough contrast to be seen against the background colour. Uses #mininum_brightness_diff and #minimum_color_diff.

Color::Palette::MonoContrast#brightness_diff(colour1, colour2)
Returns the absolute difference between the brightness levels of two colours. This will be a decimal value between 0 and 1. Accessibility guidelines from the W3C for colour contrast suggest that this value be at least approximately 0.49 (125 / 255.0) for proper contrast.

Color::Palette::MonoContrast#color_diff(colour1, colour2)
Returns the contrast between to colours, a decimal value between 0 and 3. Accessibility guidelines from the W3C for colour contrast suggest that this value be at least approximately 1.96 (500 / 255.0) for proper contrast.

Color::Palette::Gimp
A class that can read a Gimp (GNU Image Manipulation Program) palette file and provide a Hash-like interface to the contents. Gimp colour palettes are RGB values only.

Because two or more entries in a Gimp palette may have the same name, all named entries are returned as an array.

```
pal = Color::Palette::Gimp.from_file(my_gimp_palette)
pal[0]          => Color::RGB<...>
pal["white"]    => [ Color::RGB<...> ]
pal["unknown"]  => [ Color::RGB<...>, Color::RGB<...>, ... ]
```

Gimp Palettes are always indexable by insertion order (an integer key).

Color::Palette::Gimp.new(palette)
Iterates through the data of the Gimp palette and interprets it.

Color::Palette::Gimp.from_file(filename)
Reads the palette data from the filename.

Color::Palette::Gimp.from_io(io)
Reads the palette data from the IO or IO-like object (depends on #read).

Color::Palette::Gimp#[](key)
Access the colours by name or numeric index.

Color::Palette::Gimp#each
Loops through the palette’s colours, yielding each colour in turn.

Color::Palette::Gimp#each_name
Loops through the named colours, yielding the colour name and an Array of colours.

Color::Palette::Gimp#valid?
Returns true if this is believed to be a valid GIMP palette.

Color::Palette::Gimp#name
Returns the name of the GIMP palette.

12. PDF::Writer Demos

A quick overview of the demo programs and what they are expected to do. These may be downloaded separately from the main PDF::Writer package at the RubyForge project for PDF::Writer.

If PDF::Writer has been installed with RubyGems, then the demos will need to be run explicitly referencing RubyGems:

```
% ruby -rubygems chunkybacon.rb
```

chunkybacon.rb

Creates a single-page document with three copies of an image from Why's (Poignant) Guide to Ruby.

code.rb

An example of a custom markup tag. This code is not included in the main release because it doesn't completely work.

demo.rb

Shows how gradient colours can be generated and how text can be rotated around a common point.

gettysburg.rb

Wraps the Gettysburg Address in an automatically sized rounded rectangle.

hello.rb

A simple "hello, world" sort of demo.

individual-i.rb

Provides a class to generate an Indivdual I and then generates a lot of them in different colours and sizes.

pac.rb

A riff on an old arcade game.

pagenumber.rb

A demonstration of a complex page numbering scheme. Uses a very small page size (A10) for easy visibility.

qr-language.rb

Creates a QuickRef sheet for the Ruby language. Based on ZenSpider's original text.

qr-library.rb

Creates a QuickRef sheet for the Ruby standard library. Based on ZenSpider's original text.

13. Future Plans

There are great plans for future versions of PDF::Writer.

- Support for Text::Hyphen hyphenation in text wrapping methods.
- Widow/orphan support (so that paragraphs never have less than two lines on a page).
- First line indent, hanging indent.
- More chart types: pie chart, bar chart, line chart.
- Sparklines (inline, text-sized charts, first described by Edward Tufte).
- Drop-caps.
- In-line font changes, such as <c:code>.
- Slideshows (both plain PDF and SVG).
- Event detection: callbacks on page change, etc. This will be necessary to support multiple page sizes and orientations within a single document.
- Support for code page maps; these are predefined encodings and encoding difference sets.
- Unicode support—at least UTF-16.

14. Revision History

PDF::Writer 1.1.3: September 9, 2005

- Fixed #2356 submitted by Matthew Thill. Margins set by the various margins methods would behave incorrectly.

PDF::Writer 1.1.2: August 25, 2005

- Thomas Gantner <thomas.gantner@gmx.net> found a problem with the interpretation of the placement of page numbers and provided a patch. Thanks!
- Thomas also reported a problem with an inability to place page numbering on the first page of a document, as well as strange results from not providing a starting page number. This has been fixed. Also reported as #2204.
- Modified PDF::Writer requirements to require color-tools version 1.3.0 or later. This necessitates that Color constants be changed to Color::RGB constants.
- Updated supporting library documentation to include information on color-tools version 1.3.
- Fixed a bug with Graphics#transform_matrix truncating one of the transform parameters.
- Applied a fix to PDF::SimpleTable submitted by Mike Leddy in #2129 on the RubyForge tracker.
- Applied a partial fix for PNG with index-transparency provided by Marc Vleugels in #2245 on the RubyForge tracker. NOTE: This does not solve the transparency problem; at this point, PDF::Writer cannot be said to support PNGs with transparency. This will be fixed in a later version of PDF::Writer.

Version 1.1.1: July 1, 2005

- Fixed the packaging process; the .tar.gz files will no longer corrupt the images.
- Added the images and the manual (both raw and generated) to the demo package.

Version 1.1.0: June 29, 2005

NOTE: The first two changes are incompatible with previous versions of PDF::Writer. A 90º angle in the PDF::Writer 1.0.x must be represented as a -90º (or 270º) angle in PDF::Writer 1.1 or later.

- Axis transformations in PDF::Writer::Graphics have been fixed.
- Text axis transformation in PDF::Writer#add_text has been fixed.
- Changed #text_width and #text_line_width so that the text value is the first parameter and the size parameter is second and optional. The code warns about it now, but it will break in PDF::Writer 2.0.
- Changed #add_text and #add_text_wrap so that the text parameter is before the now-optional size parameter. The code warns about it now, but it will break in PDF::Writer 2.0.
- Added #transform_matrix.
- Fixed compression. NOTE: Compression must be set early in the documentation process, or only some items will be compressed in the document. The various #save_as methods have been changed to reflect this fact.
- Enabled the placement of encoding differences dictionaries in the resulting PDF document. This change should be considered experimental.
- Added TTF licence checking. The embedding of a file not licenced for inclusion in a document will continue, but a warning will be output to standard error. This code has been gakked from FPDF (http://www.fpdf.org).
- Properly supporting symbolic font flags now.

- Added the ability to make images clickable links with any of the three image insertion methods.

Version 1.0.1: June 13, 2005

- Fixed a few minor gem issues.
- Renamed bin/manual to bin/techbook.
- Fixed the manual.pwd locator for the default install.
- Added support and documentation for a separately downloadable demo package.
- Expanded the installation documentation.

Version 1.0.0: June 12, 2005

- First production-ready release of PDF::Writer. Dozens of bug fixes, new features, and a major rewrite of the API.
- Integrated ezwriter.rb functionality with writer.rb.
- Factored out some functionality into modules and classes.
- Added CMYK colour support to JPEG images.
- Uses Color::CMYK (from color-utils) to provide CMYK support to drawing capabilities.
- Simplified the image API significantly.
- Modified image support to properly handle full image flipping.
- Fixed several multi-column issues.
- Fixed a bug where new pages automatically started by overflowing text may have resulted in writing the first line of text above the top margin. This may need further testing and probably causes problems with column handling.
- Fixed some page numbering issues.
- Added several demos, including Ruby Quick Reference Sheets.
- Changed installer to setup.rb 3.3.1-modified.
- Applied an image resize patch; the image will be resized manually before checking to see if a new page is required. Thanks to Leslie Hensley.
- Fixed a bug where entities would generate the width specified for the component characters, not the represented characters.
- Fixed a performance bug. Thanks again to Leslie Hensley.

Version 0.1.2: CVS only

- Fixed a problem with the improper reading of character numbers from .afm files that are not default files. Other font issues remain but will be fixed at a later date.

Version 0.1.0: September, 2003

- Initial technology-preview release of PDF::Writer, based on version 009e of the PHP cPDF class by R&OS.

15. Licence

PDF::Writer for Ruby

Works Included Under Other Licences

pdf/writer/graphics/imageinfo.rb

PDF::Writer includes a derivative of Keisuke Minami's ImageSize library, which can be found at rubycgi. This work—and only this or other named works—may be treated as under the Ruby licensing scheme (GPL 2 or later, Ruby's licence) as well as the overall PDF::Writer licence.

Ruby Quick Reference Sheets

PDF::Writer has demo programs that will create Quick Reference cards for Ruby. The content and output of these programs is copyright 2003–2005 Ryan Davis and is licensed under the Creative Commons Attribution Noncommercial Share Alike licence.

Images from "Why's (Poignant) Guide to Ruby"

One of the demo programs uses images originally from "Why's (Poignant) Guide to Ruby", with permission. These immages are released under the Creative Commons Attributions ShareAlike licence.

Adobe PostScript AFM Files

The file "pdf/writer/fonts/MustRead.html" and the 14 PostScript® AFM files it accompanies may be used, copied, and distributed for any purpose and without charge, with or without modification, provided that all copyright notices are retained; that the AFM files are not distributed without this file; that all modifications to this file or any of the AFM files are prominently noted in the modified file(s); and that this paragraph is not modified. Adobe Systems has no responsibility or obligation to

support the use of the AFM files.

Other Credits

R & OS PDF Class for PDF

PDF::Writer is based originally on the R & OS PDF class for PHP, which is released as public domain.

Standard Deviation Chart

The standard deviation chart (pdf/charts/stddev.rb) class is based on work by Cris Ewing of the University of Washington School of Medicine, originally created for the R & OS PDF class for PHP. He has graciously donated the code for PDF::Writer for Ruby.

bluesmoke.jpg

The logo image for PDF::Writer, bluesmoke.jpg, is modified from a picture taken by Matthew "TheSaint" Bowden and is available on the stock.xchng® at http://www.sxc.hu/browse.phtml?f=view&id=275430. Many thanks to Matthew for the use of this image.

Patents Covering the Adobe PDF Format

ADOBE PATENTS

This software is based on Adobe's PDF Reference, Third Edition, version 1.6. There may be limitations on the use of this library based on patent restrictions from Adobe. See "Patent Clarification Notice: Reading and Writing PDF Files" for more information.

UNISYS LZW PATENT

This software does not fully conform to the Adobe PDF specification because no support for LZW is included in this software. At the time of the initial development of this software (2003), the Unisys LZW patent was still in effect outside of the United States. LZW support will be added in a later version of PDF::Writer.

Patent Clarification Notice: Reading and Writing PDF Files

Adobe has a number of patents covering technology that is disclosed in the Portable Document Format (PDF) Specification, version 1.6 and later, as documented in PDF Reference and associated Technical Notes (the "Specification"). Adobe desires to promote the use of PDF for information interchange among diverse products and applications.

Accordingly, the following patents are licensed on a royalty-free, non-exclusive basis for the term of each patent and for the sole purpose of developing software that produces, consumes, and interprets PDF files that are compliant with the Specification:

U.S. Patent Numbers:

- 5,634,064
- 5,737,599
- 5,781,785
- 5,819,301
- 6,028,583
- 6,289,364
- 6,421,460

In addition, the following patent is licensed on a royalty-free, non-exclusive basis for its term and for the sole purpose of developing software that produces PDF files that are compliant with the Specification (specifically excluding, however, software that consumes and/or interprets PDF files):

U.S. Patent Numbers:

- 5,860,074

The above licenses are limited to only those rights required to implement the Specification and no others. That is to say, Adobe grants only those rights in the above patent(s) necessarily practiced to implement the Specification, and does not grant any rights not required to implement the Specification. The licenses do not grant the right to practice any patent covering other technologies, such as implementation techniques that are not explicitly disclosed in the Specification, nor does it allow the use of any patented feature for any purpose other than as set forth in the applicable license grant. Adobe has other patents in various fields, none of which are hereby licensed.