Prawn

by example

Last Update: 2015-07-15
Prawn Version: 2.0.2
git commit: e33f178

How to read this manual

This manual is a collection of examples categorized by theme and organized from the least to the most complex. While it covers most of the common use cases it is not a comprehensive guide.

The best way to read it depends on your previous knowledge of Prawn and what you need to accomplish.

If you are beginning with Prawn the first chapter will teach you the most basic concepts and how to create pdf documents. For an overview of the other features each chapter beyond the first either has a Basics section (which offer enough insight on the feature without showing all the advanced stuff you might never use) or is simple enough with only a few examples.

Once you understand the basics you might want to come back to this manual looking for examples that accomplish tasks you need.

Advanced users are encouraged to go beyond this manual and read the source code directly if any doubt is not directly covered on this manual.

Reading the examples

The title of each example is the relative path from the Prawn source manual/ folder.

The first body of text is the introductory text for the example. Generally it is a short description of the features illustrated by the example.

Next comes the example source code block in fixed width font.

Most of the example snippets illustrate features that alter the page in place. The effect of these snippets is shown right below a dashed line. If it doesn't make sense to evaluate the snippet inline, a box with the link for the example file is shown instead.

Note that the `stroke_axis` method used throughout the manual is part of standard Prawn. It is defined in this file:

https://github.com/prawnpdf/prawn/blob/master/lib/prawn/graphics.rb

Basic concepts

This chapter covers the minimum amount of functionality you'll need to start using Prawn.

If you are new to Prawn this is the first chapter to read. Once you are comfortable with the concepts shown here you might want to check the Basics section of the Graphics, Bounding Box and Text sections.

The examples show:

- How to create new pdf documents in every possible way
- Where the origin for the document coordinates is. What are Bounding Boxes and how they interact with the origin
- How the cursor behaves
- How to start new pages
- What the base unit for measurement and coordinates is and how to use other convenient measures
- How to build custom view objects that use Prawn's DSL

There are three ways to create a PDF Document in Prawn: creating a new **Prawn::Document** instance, or using the **Prawn::Document.generate** method with and without block arguments.

The following snippet showcase each way by creating a simple document with some text drawn.

When we instantiate the **Prawn::Document** object the actual pdf document will only be created after we call **render_file**.

The generate method will render the actual pdf object after exiting the block. When we use it without a block argument the provided block is evaluated in the context of a newly created **Prawn::Document** instance. When we use it with a block argument a **Prawn::Document** instance is created and passed to the block.

The generate method without block arguments requires less typing and defines and renders the pdf document in one shot. Almost all of the examples are coded this way.

Assignment Implicit Block Explicit Block

```
# Assignment
pdf = Prawn::Document.new
pdf.text "Hello World"
pdf.render_file "assignment.pdf"

# Implicit Block
Prawn::Document.generate("implicit.pdf") do
  text "Hello World"
end

# Explicit Block
Prawn::Document.generate("explicit.pdf") do |pdf|
  pdf.text "Hello World"
end
```

This code snippet was not evaluated inline. You may see its output by running the example file located here:
http://github.com/prawnpdf/prawn/tree/master/manual/basic_concepts/creation.rb

This is the most important concept you need to learn about Prawn:

PDF documents have the origin `[0,0]` at the bottom-left corner of the page.

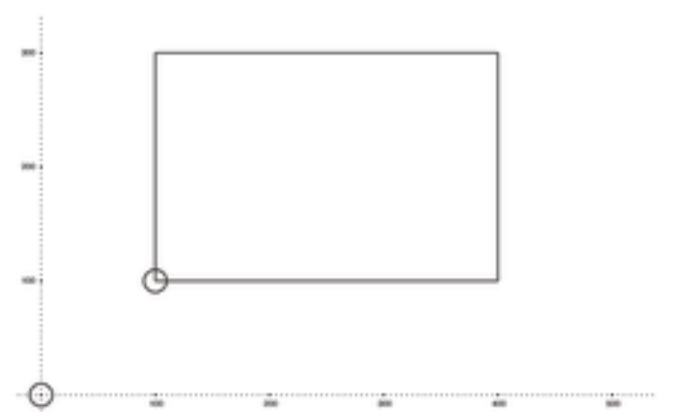
A bounding box is a structure which provides boundaries for inserting content. A bounding box also has the property of relocating the origin to its relative bottom-left corner. However, be aware that the location specified when creating a bounding box is its top-left corner, not bottom-left (hence the `[100, 300]` coordinates below).

Even if you never create a bounding box explicitly, each document already comes with one called the margin box. This initial bounding box is the one responsible for the document margins.

So practically speaking the origin of a page on a default generated document isn't the absolute bottom left corner but the bottom left corner of the margin box.

The following snippet strokes a circle on the margin box origin. Then strokes the boundaries of a bounding box and a circle on its origin.

```
stroke_axis

stroke_circle [0, 0], 10

bounding_box[100, 300], width: 300, height: 200 do
  stroke_bounds
  stroke_circle [0, 0], 10
end
```

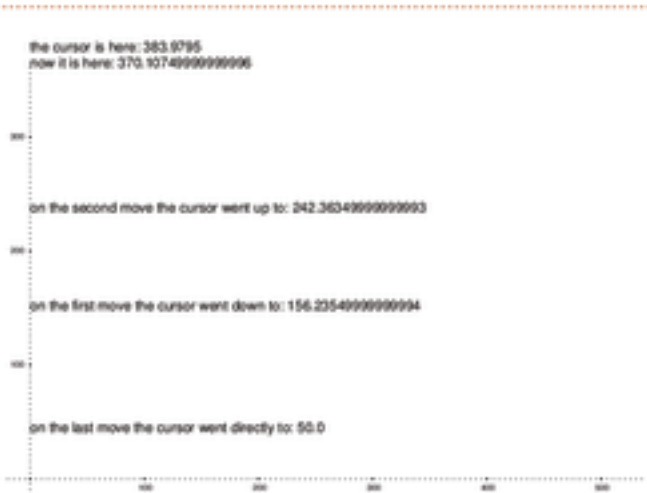


We normally write our documents from top to bottom and it is no different with Prawn. Even if the origin is on the bottom left corner we still fill the page from the top to the bottom. In other words the cursor for inserting content starts on the top of the page.

Most of the functions that insert content on the page will start at the current cursor position and proceed to the bottom of the page.

The following snippet shows how the cursor behaves when we add some text to the page and demonstrates some of the helpers to manage the cursor position. The `cursor` method returns the current cursor position.

```
stroke_axis

text "the cursor is here: #{cursor}"
text "now it is here: #{cursor}"

move_down 200
text "on the first move the cursor went down to: #{cursor}"

move_up 100
text "on the second move the cursor went up to: #{cursor}"

move_cursor_to 50
text "on the last move the cursor went directly to: #{cursor}"
```



Another group of helpers for changing the cursor position are the pad methods. They accept a numeric value and a block. **pad** will use the numeric value to move the cursor down both before and after the block content. **pad_top** will only move the cursor before the block while **pad_bottom** will only move after.

float is a method for not changing the cursor. Pass it a block and the cursor will remain on the same place when the block returns.

```
stroke_horizontal_rule
pad(20) { text "Text padded both before and after." }

stroke_horizontal_rule
pad_top(20) { text "Text padded on the top." }

stroke_horizontal_rule
pad_bottom(20) { text "Text padded on the bottom." }

stroke_horizontal_rule
move_down 30

text "Text written before the float block."

float do
  move_down 30
  bounding_box([0, cursor], :width => 200) do
    text "Text written inside the float block."
    stroke_bounds
  end
end

text "Text written after the float block."
```

Text padded both before and after.

Text padded on the top.

Text padded on the bottom.

Text written before the float block.
Text written after the float block.

Text written inside the float block.

A PDF document is a collection of pages. When we create a new document be it with **Document.new** or on a **Document.generate** block one initial page is created for us.

Some methods might create new pages automatically like **text** which will create a new page whenever the text string cannot fit on the current page.

But what if you want to go to the next page by yourself? That is easy.

Just use the **start_new_page** method and a shiny new page will be created for you just like in the following snippet.

```
text "We are still on the initial page for this example. Now I'll ask " +
     "Prawn to gently start a new page. Please follow me to the next page."

start_new_page

text "See. We've left the previous page behind."
```

We are still on the initial page for this example. Now I'll ask Prawn to gently start a new page. Please follow me to the next page.

See. We've left the previous page behind.

The base unit in Prawn is the PDF Point. One PDF Point is equal to 1/72 of an inch.

There is no need to waste time converting this measure. Prawn provides helpers for converting from other measurements to PDF Points.

Just `require "prawn/measurement_extensions"` and it will mix some helpers onto `Numeric` for converting common measurement units to PDF Points.

```
require "prawn/measurement_extensions"

[:mm, :cm, :dm, :m, :in, :yd, :ft].each do |measurement|
  text "1 #{measurement} in PDF Points: #{1.send(measurement)} pt"
  move_down 5.mm
end
```

1 mm in PDF Points: 2.8346456692913391 pt

1 cm in PDF Points: 28.346456692913391 pt

1 dm in PDF Points: 283.46456692913387 pt

1 m in PDF Points: 2834.6456692913391 pt

1 in in PDF Points: 72 pt

1 yd in PDF Points: 2592 pt

1 ft in PDF Points: 864 pt

To create a custom class that extends Prawn's functionality, use the `Prawn::View` mixin. This approach is safer than creating subclasses of `Prawn::Document`s while being just as convenient.

By using this mixin, your state will be kept completely separate from `Prawn::Document`'s state, and you will avoid accidental method collisions within `Prawn::Document`.

To build custom classes that make use of other custom classes, you can define a method named `document()` that returns any object that acts similar to a `Prawn::Document` object. `Prawn::View` will then direct all delegated calls to that object instead.

```
class Greeter
  include Prawn::View

  def initialize(name)
    @name = name
  end

  def say_hello
    text "Hello, #{@name}!"
  end

  def say_goodbye
    font("Courier") do
      text "Goodbye, #{@name}!"
    end
  end
end

greeter = Greeter.new("Gregory")

greeter.say_hello
greeter.say_goodbye

greeter.save_as("greetings.pdf")
```

This code snippet was not evaluated inline. You may see its output by running the example file located here:
http://github.com/prawnpdf/prawn/tree/master/manual/basic_concepts/view.rb

Graphics

Here we show all the drawing methods provided by Prawn. Use them to draw the most beautiful imaginable things.

Most of the content that you'll add to your pdf document will use the graphics package. Even text is rendered on a page just like a rectangle so even if you never use any of the shapes described here you should at least read the basic examples.

The examples show:

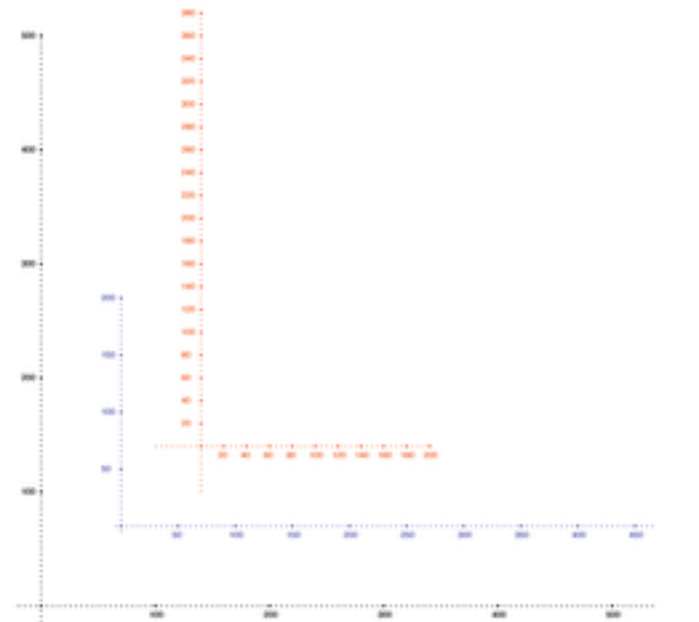
- All the possible ways that you can fill or stroke shapes on a page
- How to draw all the shapes that Prawn has to offer from a measly line to a mighty polygon or ellipse
- The configuration options for stroking lines and filling shapes
- How to apply transformations to your drawing space

To produce this manual we use the **stroke_axis** helper method within the examples.

stroke_axis prints the *x* and *y* axis for the current bounding box with markers in 100 increments. The defaults can be changed with various options.

Note that the examples define a custom **:height** option so that only the example canvas is used (as seen with the output of the first line of the example code).

```
stroke_axis
stroke_axis(at => [70, 70], :height => 100, :step_length => 50,
            :negative_axes_length => 5, :color => '000000')
stroke_axis(at => [140, 140], :width => 200, :height => curve.to_i - 140,
            :step_length => 20, :negative_axes_length => 40, :color => '0000ff')
```



graphics/fill_and_stroke.rb

There are two drawing primitives in Prawn: `fill` and `stroke`.

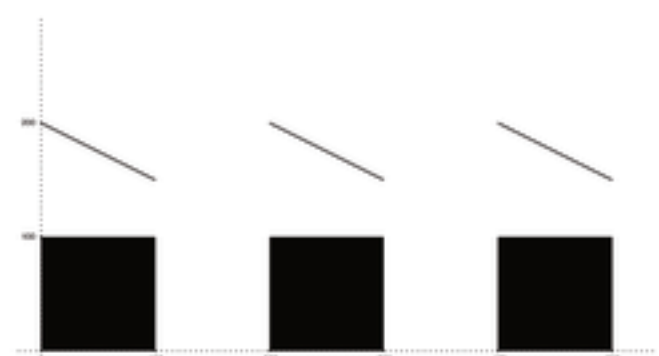
These are the methods that actually draw stuff on the document. All the other drawing shapes like `rectangle`, `circle` or `line_to` define drawing paths. These paths need to be either stroked or filled to gain form on the document.

Calling these methods without a block will act on the drawing path that has been defined prior to the call.

Calling with a block will act on the drawing path set within the block.

Most of the methods which define drawing paths have methods of the same name starting with stroke_ and fill_ which create the drawing path and then stroke or fill it.

```
stroke_axis

# No block
line [0, 200], [100, 150]
stroke

rectangle [0, 100], 100, 100
fill

# With block
stroke { line [200, 200], [300, 150] }
fill   { rectangle [200, 100], 100, 100 }

# Method hook
stroke_line [400, 200], [500, 150]
fill_rectangle [400, 100], 100, 100
```



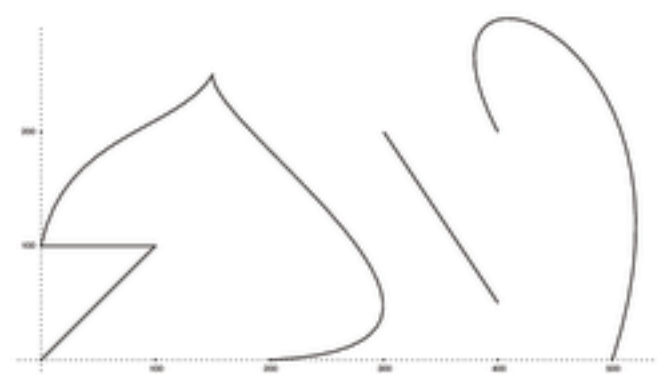
Prawn supports drawing both lines and curves starting either at the current position, or from a specified starting position.

`line_to` and `curve_to` set the drawing path from the current drawing position to the specified point. The initial drawing position can be set with `move_to`. They are useful when you want to chain successive calls because the drawing position will be set to the specified point afterwards.

`line` and `curve` set the drawing path between the two specified points.

Both curve methods define a Bezier curve bounded by two additional points provided as the `:bounds` param.

```
stroke_axis

# line_to and curve_to
stroke do
  move_to 0, 0

  line_to 100, 100
  line_to 0, 100

  curve_to [150, 250], :bounds => [[125, 200], [125, 200]]
  curve_to [200, 0],   :bounds => [[150, 200], [450, 10]]
end

# line and curve
stroke do
  line [300, 200], [400, 50]
  curve [500, 0], [400, 200], :bounds => [[400, 300], [300, 300]]
end
```



graphics/common_lines.rb

Prawn provides helpers for drawing some commonly used lines:

`vertical_line` and **`horizontal_line`** do just what their names imply. Specify the start and end point at a fixed coordinate to define the line.

`horizontal_rule` draws a horizontal line on the current bounding box from border to border, using the current y position.

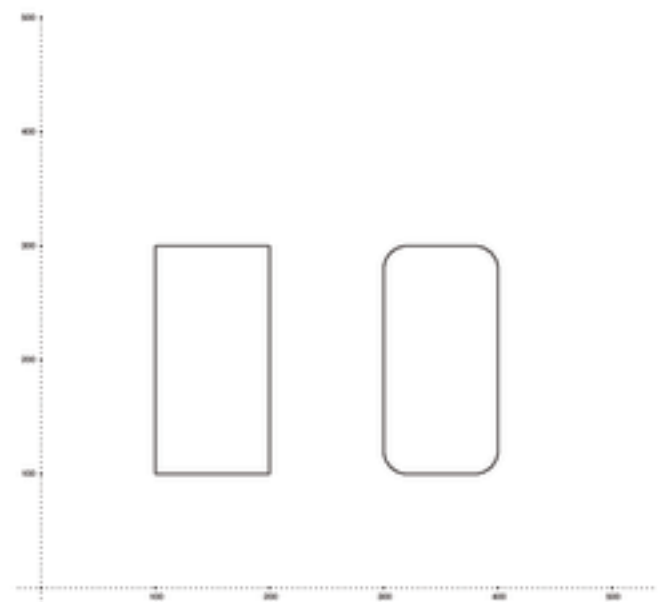
```
stroke_axis

stroke_color "FF6600"

stroke do
  # just lower the current y position
  move_down 50
  horizontal_rule

  vertical_line  100, 300, :at => 50

  horizontal_line 200, 500, :at => 150
end
```

graphics/rectangle.rb

To draw a rectangle, just provide the upper-left corner, width and height to the `rectangle` method.

There's also `rounded_rectangle`. Just provide an additional radius value for the rounded corners.

```
stroke_axis

stroke do
  rectangle [100, 300], 100, 200

  rounded_rectangle [300, 300], 100, 200, 20
end
```

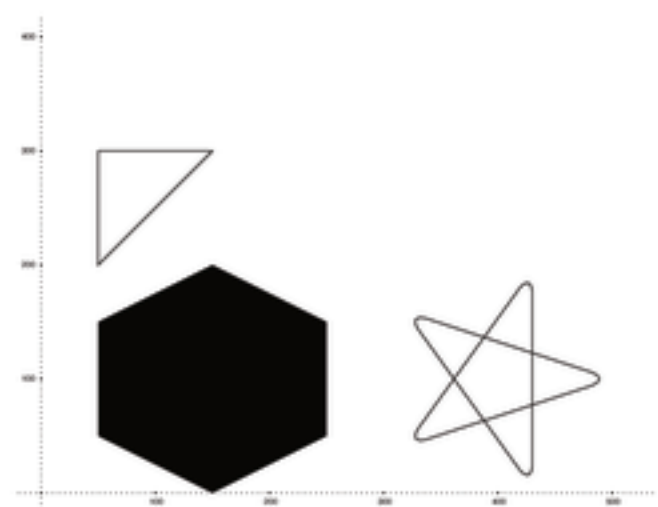


graphics/polygon.rb

Drawing polygons in Prawn is easy, just pass a sequence of points to one of the polygon family of methods.

Just like `rounded_rectangle` we also have `rounded_polygon`. The only difference is the radius param comes before the polygon points.

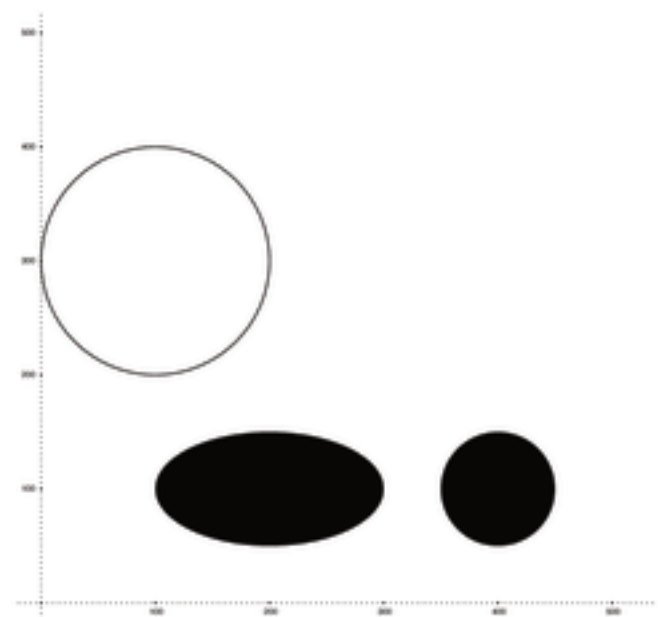
```
stroke_axis

# triangle
stroke_polygon [50, 200], [50, 300], [150, 300]

# hexagon
fill_polygon [50, 150], [100, 200], [150, 150],
             [150, 50], [100, 0], [50, 50]

# pentagon
pentagon_points = [500, 100], [400, 5], [319, 41], [319, 159], [419, 195]
pentagon_points = [0, 2, 4, 1, 3].map{ |i| pentagon_points[i] }

stroke_rounded_polygon(20, *pentagon_points)
```



To define a `circle` all you need is the center point and the radius.

To define an `ellipse` you provide the center point and two radii (or axes) values. If the second radius value is ommited, both radii will be equal and you will end up drawing a circle.

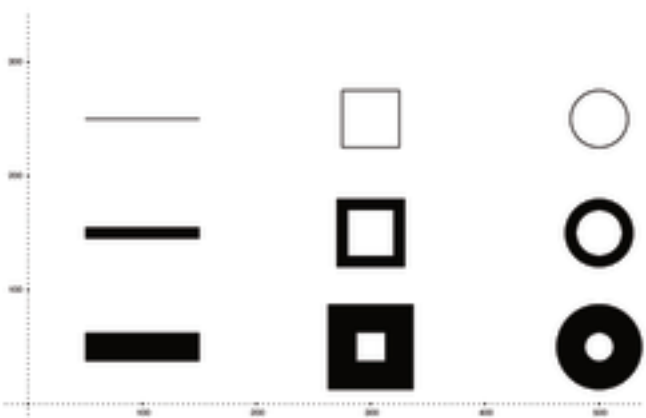
```
stroke_axis

stroke_circle [100, 300], 100

fill_ellipse [200, 100], 100, 50

fill_ellipse [400, 100], 50
```



The `line_width=` method sets the stroke width for subsequent stroke calls.

Since Ruby assumes that an unknown variable on the left hand side of an assignment is a local temporary, rather than a setter method, if you are using the block call to `Prawn::Document.generate` without passing params you will need to call `line_width` on self.

```
stroke_axis

y = 230

3.times do |i|
  case i
  when 0 then line_width = 10       # this call will have no effect
  when 1 then self.line_width = 10
  when 2 then self.line_width = 25
  end

  stroke do
    horizontal_line 50, 150, :at => y
    rectangle [275, y - 25], 50, 50
    circle [500, y], 25
  end

  y -= 100
end
```

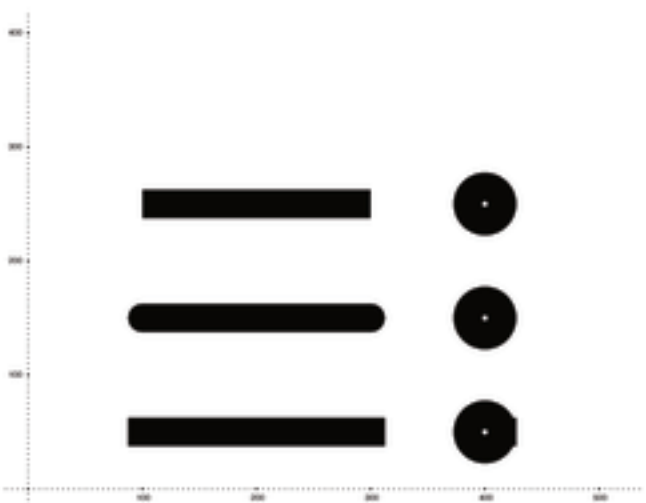


The cap style defines how the edge of a line or curve will be drawn. There are three types: `:butt` (the default), `:round` and `:projecting_square`.

The difference is better seen with thicker lines. With `:butt` lines are drawn starting and ending at the exact points provided. With both `:round` and `:projecting_square` the line is projected beyond the start and end points.

Just like `line_width=` the `cap_style=` method needs an explicit receiver to work.

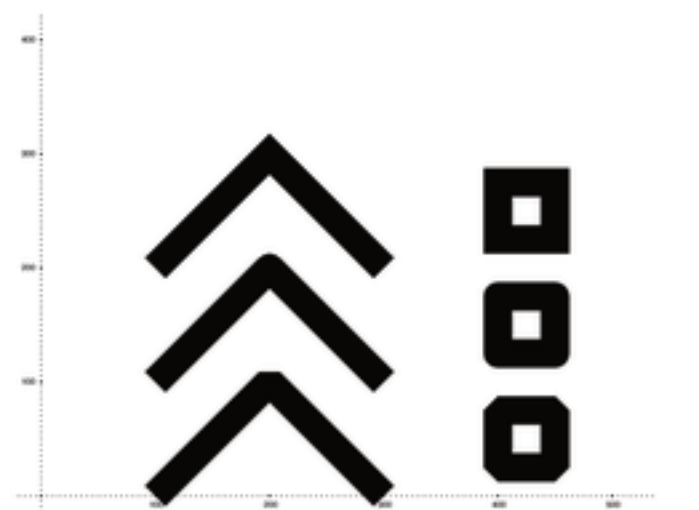
```
stroke_axis

self.line_width = 25

[:butt, :round, :projecting_square].each_with_index do |cap, i|
  self.cap_style = cap

  y = 250 - i * 100
  stroke_horizontal_line 100, 300, at: y
  stroke_circle [400, y], 15
end
```



The join style defines how the intersection between two lines is drawn. There are three types: `:miter` (the default), `:round` and `:bevel`.

Just like `cap_style`, the difference between styles is better seen with thicker lines.

```
stroke_axis

self.line_width = 25

[:miter, :round, :bevel].each_with_index do |style, i|
  self.join_style = style

  y = 200 - i * 100
  stroke do
    move_to(0,0, y)
    line_to(200, y - 100)
    line_to(300, y)
  end
  stroke_rectangle [400, y + 75], 50, 50
end
```



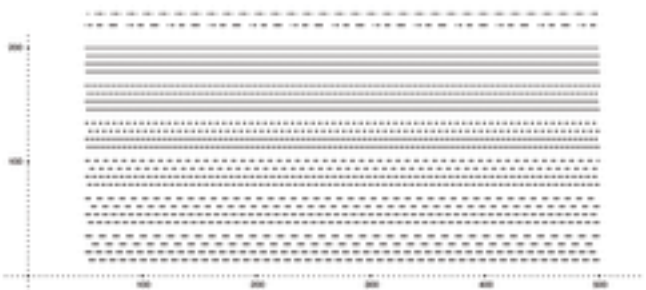
This sets the dashed pattern for lines and curves. The (dash) length defines how long each dash will be.

The **:space** option defines the length of the space between the dashes.

The **:phase** option defines the start point of the sequence of dashes and spaces.

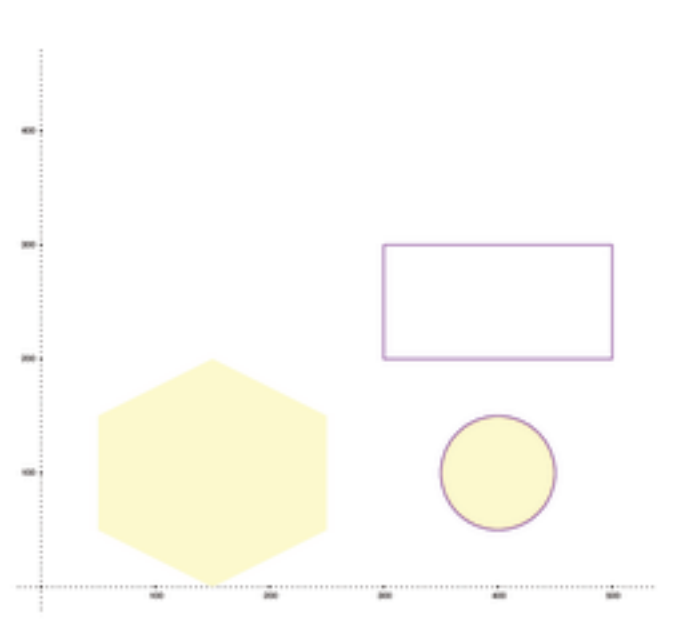
Complex dash patterns can be specified by using an array with alternating dash/gap lengths for the first parameter (note that the **:space** option is ignored in this case).

```
stroke_axis

dash(3, 2, 3, 4, 1, 5), :phase => 0)
stroke_horizontal_line 50, 500, :at => 230
dash(3, 2, 3, 4, 5, 6, 7, 8))
stroke_horizontal_line 50, 500, :at => 220

base_y = 210

24.times do |i|
  length = (i / 4) + 1
  space = length            # space between dashes same length as dash
  phase = 0                 # start with dash

  case i % 4
  when 0 then base_y -= 5
  when 1 then phase = length        # start with space between dashes
  when 2 then space = length * 0.5  # space between dashes half as long as dash
  when 3
    space = length * 0.5    # space between dashes half as long as dash
    phase = length          # start with space between dashes
  end
  base_y -= 5

  dash(length, :space => space, :phase => phase)
  stroke_horizontal_line 50, 500, :at => base_y - (i * 2)
end
```



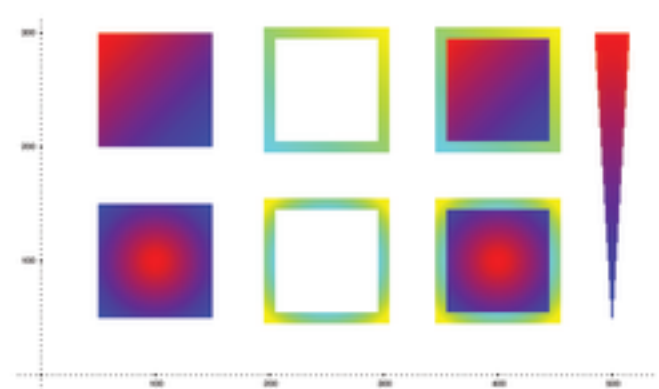
We can change the stroke and fill colors providing an HTML rgb 6 digit color code string (“AB1234”) or 4 values for CMYK.

```
stroke_axis

# Fill with hollow using HEX
fill_color "FFFFCC"
fill_polygon [50, 150], [150, 200], [250, 150],
             [250, 50], [150, 0], [50, 50]

# Stroke with Purple using CMYK
stroke_color 50, 100, 0, 0
stroke_rectangle [300, 300], 200, 100

# Both together
fill_and_stroke_circle [400, 100], 50
```



Note that because of the way PDF renders radial gradients in order to get solid fill your start circle must be fully inside your end circle. Otherwise you will get triangle fill like illustrated in the example below.

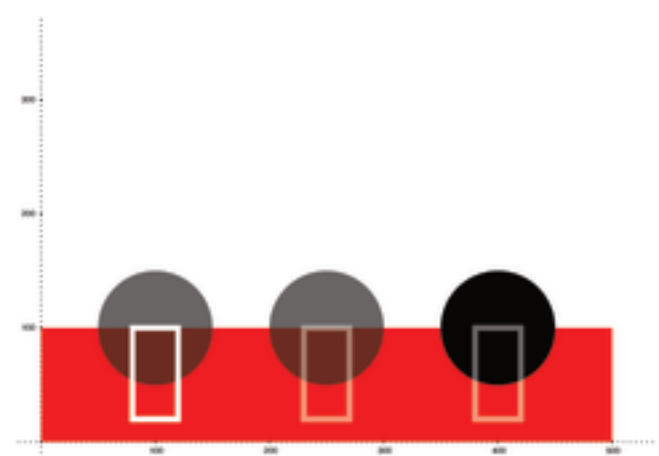
```
stroke_axis
self.line_width = 10

fill_gradient [50, 300], [150, 200], 'FF0000', '000088'
fill_rectangle [50, 300], 100, 100

stroke_gradient [200, 200], [300, 300], '000088', 'FF0000'
stroke_rectangle [200, 300], 100, 100

fill_gradient [350, 300], [450, 200], 'FF0000', '000088'
stroke_gradient [350, 200], [450, 300], '000088', 'FF0000'
fill_and_stroke_rectangle [350, 300], 100, 100

fill_gradient [100, 100], 0, [100, 100], 70.7%, 'FF0000', '000088'
fill_rectangle [50, 150], 100, 100

stroke_gradient [250, 100], 45, [250, 100], 70.7%, '000088', 'FF0000'
stroke_rectangle [200, 150], 100, 100

stroke_gradient [400, 100], 45, [400, 100], 70.7%, '000088', 'FF0000'
fill_gradient [400, 100], 0, [400, 100], 70.7%, 'FF0000', '000088'
fill_and_stroke_rectangle [350, 150], 100, 100

fill_gradient [500, 300], 15, [500, 50], 0, 'FF0000', '000088'
fill_rectangle [485, 300], 30, 250
```



Although the name of the method is **transparency**, what we are actually setting is the opacity for fill and stroke. So 0 means completely transparent and `1.0` means completely opaque

You may call it providing one or two values. The first value sets fill opacity and the second value sets stroke opacity. If the second value is omitted fill and stroke will have the same opacity.

```
stroke_axis

self.line_width = 5
fill_color "FF0000"
fill_rectangle [0, 100], 500, 100

fill_color "000000"
stroke_color "FFFFFF"

base_x = 100
[0.5, 1, 0.5, [1, 0.5]].each do |arg|
  transparent(*arg) do
    fill_circle [base_x, 100], 50
    stroke_rectangle [base_x - 20, 100], 40, 80
  end

  base_x += 150
end
```



Soft masks are user for more complex alpha channel manipulations. You can use arbitrary drawing functions for creation of soft masks. The resulting alpha channel is made of greyscale version of the drawing (luminosity channel to be precise). So while you can use any combination of colors for soft masks it's easier to use greyscales. Black will result in full transparency and white will make region fully opaque.

Soft mask is a part of page graphic state. So if you want to apply soft mask only to a part of page you need to endose drawing instructions in **save_graphics_state** block.

```
save_graphics_state do
  soft_mask do
    0.upto 15 do |i|
      fill_color 0, 0, 0, 100.0 / 16.0 * (i + 1)
      fill_circle [75 + i * 25, 100], 40
    end
  end

  fill_color "009ddc"
  fill_rectangle [0, 40], 600, 20

  fill_color "40b0a0"
  fill_rectangle [0, 80], 600, 20

  fill_color "e03c3e"
  fill_rectangle [0, 100], 600, 20

  fill_color "f58220"
  fill_rectangle [0, 120], 600, 20

  fill_color "fdb827"
  fill_rectangle [0, 140], 600, 20

  fill_color "61ba46"
  fill_rectangle [0, 160], 600, 20
end
```

Prawn's fill operators (`fill` and `fill_and_stroke` both accept a `:fill_rule` option. These rules determine which parts of the page are counted as "inside" vs. "outside" the path. There are two fill rules:

* `:nonzero_winding_number` (default): a point is inside the path if a ray from that point to infinity crosses a nonzero "net number" of path segments, where path segments intersecting in one direction are counted as positive and those in the other direction negative.

* `:even_odd`: A point is inside the path if a ray from that point to infinity crosses an odd number of path segments, regardless of direction.

The differences between the fill rules only come into play with complex paths; they are identical for simple shapes.

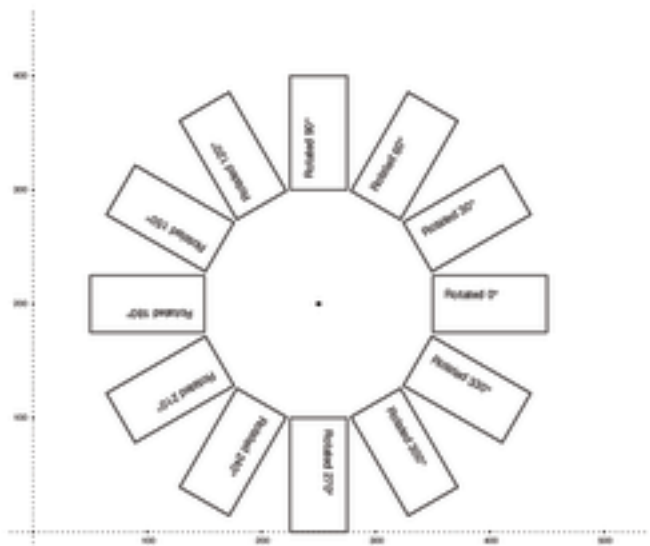
```
pentagram = [[148, 95], [6, 36], [111, 199], [111, 0], [6, 154]]

stroke_color 'FF0000'
line_width 2

text_box "Nonzero Winding Number", at => [50, 215],
                                   :width => 170,
                                   :align => :center
polygon(*pentagram.map { |x, y| [x + 50, y] })
fill_and_stroke

text_box "Even-Odd", at => [330, 215], :width => 170, :align => :center
polygon(*pentagram.map { |x, y| [x + 330, y] })
fill_and_stroke(:fill_rule => :even_odd)
```

This transformation is used to rotate the user space. Give it an angle and an `:origin` point about which to rotate and a block. Everything inside the block will be drawn with the rotated coordinates.

The angle is in degrees.

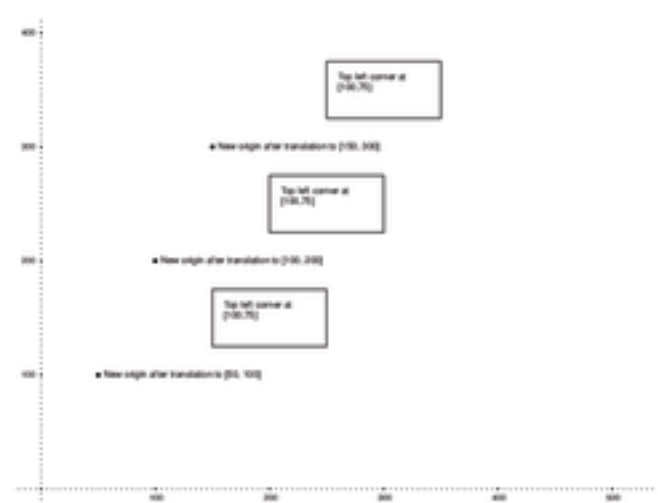
If you omit the `:origin` option the page origin will be used.

```
stroke_axis

fill_circle [250, 200], 2

12.times do |i|
  rotate(i * 30, :origin => [250, 200]) do
    stroke_rectangle [250, 225], 100, 50
    draw_text "Rotated #{i * 30}°", :size => 10, :at => [340, 205]
  end
end
```



This transformation is used to translate the user space. Just provide the x and y coordinates for the new origin.

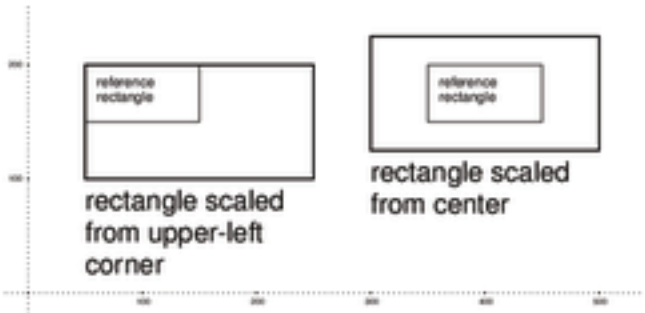
```
stroke_axis

(.upto(3) do |i|
  x = i * 50
  y = i * 100
  translate(x, y) do
    # draw a point at the new origin
    fill_circle [0, 0], 2
    draw_text "New origin after translation to [#{x}, #{y}].",
              at => [5, -2], size => 8

    stroke_rectangle [100, 75], 100, 50
    text_box "Top left corner at [100,75]",
             at => [105, 65],
             width => 90,
             size => 8
  end
end
```



This transformation is used to scale the user space. Give it an scale factor and an `:origin` point and everything inside the block will be scaled using the origin point as reference.

If you omit the `:origin` option the page origin will be used.

```
stroke_axis

width = 100
height = 50

x = 50
y = 100

stroke_rectangle [x, y], width, height
text_box "reference rectangle", :at => [x + 10, y - 10], :width => width - 20

scale(2, :origin => [x, y]) do
  stroke_rectangle [x, y], width, height
  text_box "rectangle scaled from upper-left corner",
           :at => [x, y - height - 5],
           :width => width
end

x = 350

stroke_rectangle [x, y], width, height
text_box "reference rectangle", :at => [x + 10, y - 10], :width => width - 20

scale(2, :origin => [x + width / 2, y - height / 2]) do
  stroke_rectangle [x, y], width, height
  text_box "rectangle scaled from center",
           :at => [x, y - height - 5],
           :width => width
end
```



Text

This is probably the feature people will use the most. There is no shortage of options when it comes to text. You'll be hard pressed to find a use case that is not covered by one of the text methods and configurable options.

The examples show:

- Text that flows from page to page automatically starting new pages when necessary
- How to use text boxes and place them on specific positions
- What to do when a text box is too small to fit its content
- Flowing text in columns
- How to change the text style configuring font, size, alignment and many other settings
- How to style specific portions of a text with inline styling and formatted text
- How to define formatted callbacks to reuse common styling definitions
- How to use the different rendering modes available for the text methods
- How to create your custom text box extensions
- How to use external fonts on your pdfs
- What happens when rendering text in different languages

Text rendering can be as simple or as complex as you want.

This example covers the most basic method: `text`. It is meant for free flowing text. The provided string will flow according to the current bounding box width and height. It will also flow onto the next page if the bottom of the bounding box is reached.

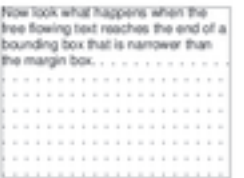
The text will start being rendered on the current cursor position. When it finishes rendering, the cursor is left directly below the text.

This example also shows text flowing across pages following the margin box and other bounding boxes.

```
move_cursor_to 50
text "This text will flow to the next page. " * 20

y_position = cursor - 50
bounding_box[0, y_position], :width => 200, :height => 150) do
  transparent(0.5) { stroke_bounds }
  text "This text will flow along this bounding box we created for it. " * 5
end

bounding_box[300, y_position], :width => 200, :height => 150) do
  transparent(0.5) { stroke_bounds }  # This will stroke on one page

  text "Now look what happens when the free flowing text reaches the end " +
       "of a bounding box that is narrower than the margin box." +
       " . " * 200 +
       "It continues on the next page as if the previous bounding box " +
       "was closed. If we want it to have the same border as the one on " +
       "the previous page we will need to stroke the boundaries again."

  transparent(0.5) { stroke_bounds }  # And this will stroke on the next
end

move_cursor_to 200
span(350, :position => :center) do
  text "Span is a different kind of bounding box as it lets the text " +
       "flow smoothly onto the next page. It doesn't matter if the text " +
       "started on the middle of the previous page, when it flows to the " +
       "next page it will start at the beginning." + " . " * 500 +
       "I told you it would start on the beginning of this page."
end
```

This text will flow to the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to

the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to the next page. This text will flow to the next page.



. . . . It continues on the next page as if the previous bounding box was cloned. If we want it to have the same border as the one on the previous page we will need to stroke the boundaries again.

Span is a different kind of bounding box as it lets the text flow gracefully onto the next page. It doesn't matter if the text started on the middle of the previous page, when it flows to the next page it will start at the beginning. _

— —
— —
— —
— —
— —
— —
— —
— —
— — — — — — — — — — — — — — — — — I told you it would start on the beginning of this page.

Sometimes we want the text on a specific position on the page. The `text` method just won't help us.

There are two other methods for this task: `draw_text` and `text_box`.

`draw_text` is very simple. It will render text starting at the position provided to the `:at` option. It won't flow to a new line even if it hits the document boundaries so it is best suited for short text.

`text_box` gives us much more control over the output. Just provide `:width` and `:height` options and the text will flow accordingly. Even if you don't provide a `:width` option the text will flow to a new line if it reaches the right border.

Given that, `text_box` is the better option available.

```
draw_text "This draw_text line is absolute positioned. However don't " +
          "expect it to flow even if it hits the document border",
          :at => [200, 300]

text_box "This is a text box, you can control where it will flow by " +
         "specifying the :height and :width options",
         :at => [100, 250],
         :height => 100,
         :width => 100

text_box "Another text box with no :width option passed, so it will " +
         "flow to a new line whenever it reaches the right margin. ",
         :at => [200, 100]
```

The `text_box` method accepts both `:width` and `:height` options. So what happens if the text doesn't fit the box?

The default behavior is to truncate the text but this can be changed with the `:overflow` option. Available modes are `:expand` (the box will increase to fit the text) and `:shrink_to_fit` (the text font size will be shrunk to fit).

If `:shrink_to_fit` mode is used with the `:min_font_size` option set, the font size will not be reduced to less than the value provided even if it means truncating some text.

If the `:disable_wrap_by_char` is set to `true` then any text wrapping done while using the `:shrink_to_fit` mode will not break up the middle of words.

```
string = "This is the sample text used for the text boxes. See how it " +
         "behave with the various overflow options used."

text string

y_position = cursor - 20
[:truncate, :expand, :shrink_to_fit].each_with_index do |mode, i|
  text_box string, at => [i * 150, y_position],
                   width => 100,
                   height => 50,
                   overflow => mode
end

string = "If the box is too small for the text, :shrink_to_fit " +
         "can render the text in a really small font size."

move_down 120
text string
y_position = cursor - 20
[nil, 8, 10, 12].each_with_index do |value, index|
  text_box string, at => [index * 150, y_position],
                   width => 50,
                   height => 50,
                   overflow => :shrink_to_fit,
                   min_font_size => value
end
```

This is the sample text used for the text boxes. See how it behave with the various overflow options used.

This is the sample text used for the text boxes. See

This is the sample text used for the text boxes. See how it behave with the various overflow options used.

This is the sample text used for the text boxes. See how it behave with the various overflow options used.

If the box is too small for the text, :shrink_to_fit can render the text in a really small font size.

If the box is too small for the text, :shrink_to_fit can render the text in a really small font size.

If the box is too small for the text, :shrink_to_fit can render

If the box is too small for the text,

If the box is too small for

Whenever the `text_box` method truncates text, this truncated bit is not lost, it is the method return value and we can take advantage of that.

We just need to take some precautions.

This example renders as much of the text as will fit in a larger font inside one text_box and then proceeds to render the remaining text in the default size in a second text_box.

```
string = "This is the beginning of the text. It will be cut somewhere and " +
         "the rest of the text will procede to be rendered this time by " +
         "calling another method." * " . " * 50

y_position = cursor - 20
excess_text = text_box string,
                      :width    => 300,
                      :height   => 50,
                      :overflow => :truncate,
                      :at       => [0.00, y_position],
                      :size     => 18

text_box excess_text,
         :width    => 300,
         :at       => [100, y_position - 100]
```

The column_box method allows you to define columns that flow their contents from one section to the next. You can have a number of columns on the page, and only when the last column overflows will a new page be created.

```
text "The Prince",           :align => :center, :size => 18
text "Niccolò Machiavelli", :align => :center, :size => 14
move_down 12

column_box([0, cursor], :columns => 2, :width => bounds.width) do
  text((<<-END.gsub(/\s+/, ' ') + "\n\n") * 3)
    All the States and Governments by which men are or ever have been ruled,
    have been and are either Republics or Princedoms. Princedoms are either
    hereditary, in which the sovereignty is derived through an ancient line
    of ancestors, or they are new. New Princedoms are either wholly new, as
    that of Milan to Francesco Sforza; or they are like limbs joined on to
    the hereditary possessions of the Prince who acquires them, as the
    Kingdom of Naples to the dominions of the King of Spain. The States thus
    acquired have either been used to live under a Prince or have been free;
    and he who acquires them does so either by his own arms or by the arms of
    others, and either by good fortune or by merit.
  END
end
```

The Prince

Niccolò Machiavelli

All the States and Governments by which men are or ever have been ruled, have been and are either Republics or Princedoms. Princedoms are either hereditary, in which the sovereignty is derived through an ancient line of ancestors, or they are new. New Princedoms are either wholly new, as that of Milan to Francesco Sforza; or they are like limbs joined on to the hereditary possessions of the Prince who acquires them, as the Kingdom of Naples to the dominions of the King of Spain. The States thus acquired have either been used to live under a Prince or have been free; and he who acquires them does so either by his own arms or by the arms of others, and either by good fortune or by merit.

All the States and Governments by which men are or ever have been ruled, have been and are either Republics or Princedoms. Princedoms are either hereditary, in which the sovereignty is derived through an ancient line of ancestors, or they are new. New Princedoms are either wholly new, as that of Milan to Francesco Sforza; or they are like limbs joined on to the hereditary possessions of the Prince who acquires them, as the Kingdom of Naples to the dominions of the King of Spain. The States thus acquired have either been used to live under a Prince or have been free; and he who acquires them does so either by his own arms or by the arms of others, and either by good fortune or by merit.

All the States and Governments by which men are or ever have been ruled, have been and are either Republics or Princedoms. Princedoms are either hereditary, in which the sovereignty is derived through an ancient line of ancestors, or they are new. New Princedoms are either wholly new, as that of Milan to Francesco Sforza; or they are like limbs joined on to the hereditary possessions of the Prince who acquires them, as the Kingdom of Naples to the dominions of the King of Spain. The States thus acquired have either been used to live under a Prince or have been free; and he who acquires them does so either by his own arms or by the arms of others, and either by good fortune or by merit.

The #font method can be used in three different ways.

If we don't pass it any arguments it will return the current font being used to render text.

If we just pass it a font name it will use that font for rendering text through the rest of the document.

It can also be used by passing a font name and a block. In this case the specified font will only be used to render text inside the block.

The default font is Helvetica.

```
text "Let's see which font we are using: #{font.inspect}"

move_down 20
font "Times-Roman"
text "Written in Times."

move_down 20
font("Courier") do
  text "Written in Courier because we are inside the block."
end

move_down 20
text "Written in Times again as we left the previous block."

move_down 20
text "Let's see which font we are using again: #{font.inspect}"

move_down 20
font "Helvetica"
text "Back to normal."
```

Let's see which font we are using: Prawn::Font::AFM< Helvetica: 12 >

Written in Times.

Written in Courier because we are inside the block.

Written in Times again as we left the previous block.

Let's see which font we are using again: Prawn::Font::AFM< Times-Roman: 12 >

Back to normal.

The `font_size` method works just like the `font` method.

In fact we can even use `font` with the `:size` option to declare which size we want.

Another way to change the font size is by supplying the `:size` option to the text methods.

The default font size is 12.

```
text "Let's see which is the current font_size: #{font_size.inspect}"

move_down 10
font_size 16
text "Yeah, something bigger!"

move_down 10
font_size(24) { text "Even bigger!" }

move_down 10
text "Back to 16 again."

move_down 10
text "Single line on 20 using the :size option.", size: 20

move_down 10
text "Back to 16 once more."

move_down 10
font("Courier", size: 10) do
  text "Yeah, using Courier 10 courtesy of the font method."
end

move_down 10
font("Helvetica", size: 12)
text "Back to normal"
```

Let's see which is the current font_size: 12

Yeah, something bigger!

Even bigger!

Back to 16 again.

Single line on 20 using the :size option.

Back to 16 once more.

Yeah, using Courier 10 courtesy of the font method.

Back to normal

Most font families come with some styles other than normal. Most common are `bold`, `italic` and `bold_italic`.

The style can be set the using the `:style` option, with either the `font` method which will set the font and style for rest of the document, or with the inline text methods.

```
["Courier", "Helvetica", "Times-Roman"].each do |example_font|
  move_down 20

  [:bold, :bold_italic, :italic, :normal].each do |style|
    font example_font, :style => style
    text "I'm writing in #{example_font} #{style}!"
  end
end
```

I'm writing in Courier (bold)
I'm writing in Courier (bold_italic)
I'm writing in Courier (italic)
I'm writing in Courier (normal)

I'm writing in Helvetica (bold)
I'm writing in Helvetica (bold_italic)
I'm writing in Helvetica (italic)
I'm writing in Helvetica (normal)

I'm writing in Times-Roman (bold)
I'm writing in Times-Roman (bold_italic)
I'm writing in Times-Roman (italic)
I'm writing in Times-Roman (normal)

text/color.rb

The `:color` attribute can give a block of text a default color, in RGB hex format or 4-value CMYK.

```
text "Default color is black"
move_down 25

text "Changed to red", color => "FF0000"
move_down 25

text "CMYK color", color => [22, 55, 70, 30]
move_down 25

text "Also works with <color rgb='0000FF'>inline</color> formatting",
     color => "000000",
     inline_format => true
```

text/alignment.rb

Horizontal text alignment can be achieved by supplying the `:align` option to the text methods. Available options are `:left` (default), `:right`, `:center`, and `:justify`.

Vertical text alignment can be achieved using the `:valign` option with the text methods. Available options are `:top` (default), `:center`, and `:bottom`.

Both forms of alignment will be evaluated in the context of the current bounding_box.

```
text "This text should be left aligned"
text "This text should be centered",      :align => :center
text "This text should be right aligned", :align => :right

bounding_box[0, 220], :width => 250, :height => 220) do
  text "This text is flowing from the left. " * 4

  move_down 15
  text "This text is flowing from the center. " * 3, :align => :center

  move_down 15
  text "This text is flowing from the right. " * 4, :align => :right

  move_down 15
  text "This text is justified. " * 4, :align => :justify
  transparent(0.5) { stroke_bounds }
end

bounding_box[300, 220], :width => 250, :height => 220) do
  text "This text should be vertically top aligned"
  text "This text should be vertically centered",      :valign => :center
  text "This text should be vertically bottom aligned", :valign => :bottom
  transparent(0.5) { stroke_bounds }
end
```

text/leading.rb

Leading is the additional space between lines of text.

The leading can be set using the `default_leading` method which applies to the rest of the document or until it is changed, or inline in the text methods with the `:leading` option.

The default leading is 0.

```
string = "Hey, what did you do with the space between my lines? " * 10
text string, :leading => 0

move_down 20
default_leading 5
text string

move_down 20
text string, :leading => 10
```

Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines?

Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines?

Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines? Hey, what did you do with the space between my lines?

Kerning is the process of adjusting the spacing between characters in a proportional font. It is usually done with specific letter pairs. We can switch it on and off if it is available with the current font. Just pass a boolean value to the `:kerning` option of the text methods.

Character Spacing is the space between characters. It can be increased or decreased and will have effect on the whole text. Just pass a number to the `:character_spacing` option from the text methods.

```
font_size(30) do
  text_box "With kerning:",    :kerning => true,  :at => [0, y - 40]
  text_box "Without kerning:", :kerning => false, :at => [0, y - 80]

  text_box "Tomato", :kerning => true,  :at => [200, y - 40]
  text_box "Tomato", :kerning => false, :at => [200, y - 80]

  text_box "WAR", :kerning => true,  :at => [400, y - 40]
  text_box "WAR", :kerning => false, :at => [400, y - 80]

  text_box "F.", :kerning => true,  :at => [500, y - 40]
  text_box "F.", :kerning => false, :at => [500, y - 80]
end

move_down 80

string = "What have you done to the space between the characters?"
[-2, -1, 0, 0.5, 1, 2].each do |spacing|
  move_down 20
  text "#{string} (character_spacing: #{spacing})",
       :character_spacing => spacing
end
```


What have you done to the space between the characters? (character_spacing: -2)

What have you done to the space between the characters? (character_spacing: -1)

What have you done to the space between the characters? (character_spacing: 0)

What have you done to the space between the characters? (character_spacing: 0.5)

What have you done to the space between the characters? (character_spacing: 1)

What have you done to the space between the characters? (character_spacing: 2)

text/paragraph_indentation.rb

Prawn strips all whitespace from the beginning and the end of strings so there are two ways to indent paragraphs:

One is to use non-breaking spaces which Prawn won't strip. One shortcut to using them is the `Prawn::Text::NBSP`.

The other is to use the `:indent_paragraphs` option with the text methods. Just pass a number with the space to indent the first line in each paragraph.

```
# Using non-breaking spaces
text " " * 10 + "This paragraph won't be indented. " * 10 +
     "\n#{Prawn::Text::NBSP * 10}" + "This one will with NBSP. " * 10

move_down 20
text "This paragraph will be indented. " * 10 +
     "\n" + "This one will too. " * 10,
     :indent_paragraphs => 60

move_down 20

text "FROM RIGHT TO LEFT:"
text "This paragraph will be indented. " * 10 +
     "\n" + "This one will too. " * 10,
     :indent_paragraphs => 60, :direction => :rtl
```

This paragraph won't be indented. This paragraph won't be indented. This paragraph won't be indented. This paragraph won't be indented. This paragraph won't be indented. This paragraph won't be indented. This paragraph won't be indented. This paragraph won't be indented. This paragraph won't be indented. This paragraph won't be indented.

          This one will with NBSP. This one will with NBSP. This one will with NBSP. This one will with NBSP. This one will with NBSP. This one will with NBSP. This one will with NBSP. This one will with NBSP. This one will with NBSP. This one will with NBSP.

This paragraph will be indented. This paragraph will be indented. This paragraph will be indented. This paragraph will be indented. This paragraph will be indented. This paragraph will be indented. This paragraph will be indented. This paragraph will be indented. This paragraph will be indented. This paragraph will be indented.

This one will too. This one will too. This one will too. This one will too. This one will too. This one will too. This one will too. This one will too. This one will too. This one will too.

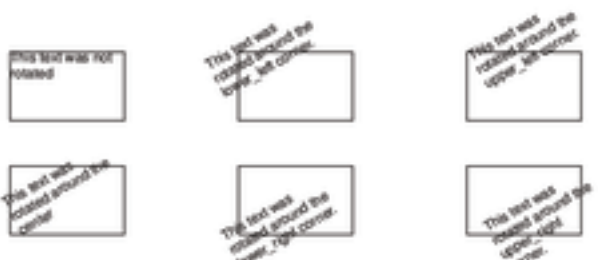
FROM RIGHT TO LEFT:

eb lliw hpargarap sihT .detnedni eb lliw hpargarap sihT .detnedni eb lliw hpargarap sihT
eb lliw hpargarap sihT .detnedni eb lliw hpargarap sihT .detnedni eb lliw hpargarap sihT .detnedni
eb lliw hpargarap sihT .detnedni eb lliw hpargarap sihT .detnedni eb lliw hpargarap sihT .detnedni
.detnedni eb lliw hpargarap sihT .detnedni
.oot lliw eno sihT .oot lliw eno sihT .oot lliw eno sihT .oot lliw eno sihT .oot lliw eno sihT
.oot lliw eno sihT .oot lliw eno sihT .oot lliw eno sihT .oot lliw eno sihT .oot lliw eno sihT

Rotating text is best avoided on free flowing text, so this example will only use the `text_box` method as we can have much more control over its output.

To rotate text all we need to do is use the `:rotate` option passing an angle in degrees and an optional `:rotate_around` to indicate the origin of the rotation (the default is `:upper_left`).

```
width  = 100
height = 60
angle  = 30
x      = 200
y      = cursor - 30

stroke_rectangle [0, y], width, height
text_box("this text was not rotated",
         :at => [0, y], :width => width, :height => height)

stroke_rectangle [0, y - 100], width, height
text_box("this text was rotated around the corner",
         :at => [0, y - 100], :width => width, :height => height,
         :rotate => angle, :rotate_around => :center)

[:lower_left, :upper_left,
 :lower_right, :upper_right].each_with_index do |corner, index|
  y = y - 100 if index == 2
  stroke_rectangle [x + (index % 2) * 200, y], width, height
  text_box("this text was rotated around the #{corner} corner.",
           :at     => [x + (index % 2) * 200, y],
           :width  => width,
           :height => height,
           :rotate => angle,
           :rotate_around => corner)
end
```



text/inline.rb

Inline formatting gives you the option to format specific portions of a text. It uses HTML-esque syntax inside the text string. Supported tags are: `b` (bold), `i` (italic), `u` (underline), `strikethrough`, `sub` (subscript), `sup` (superscript)

The following tags accept specific attributes: `font` accepts `size`, `name`, and `character_spacing`; `color` accepts `rgb` and `cmyk`; `link` accepts `href` for external links.

```
%w[b i u strikethrough sub sup].each do |tag|
  text "Just your regular text <#{tag}>except this portion</#{tag}> " +
       "is using the #{tag} tag",
       inline_format => true
  move_down 10
end

text "This <font size='18'>line</font> uses " +
     "<font name='Courier'>all the font tag</font> attributes in " +
     "<font character_spacing='2'>a single line</font>.",
     inline_format => true
move_down 10

text "Coloring in <color rgb='FF0000'>both RGB</color> " +
     "<color c='100' m='0' y='0' k='0'>and CMYK</color>",
     inline_format => true
move_down 10

text "This an external link to the " +
     "<link href='https://github.com/prawnpdf/prawn/wiki'>Prawn wiki" +
     "</link>\n",
     inline_format => true
```

Just your regular text **except this portion** is using the b tag

Just your regular text *except this portion* is using the i tag

Just your regular text except this portion is using the u tag

Just your regular text ~~except this portion~~ is using the strikethrough tag

Just your regular text $_{\text{except this portion}}$ is using the sub tag

Just your regular text $^{\text{except this portion}}$ is using the sup tag

This line uses `all the font tag` attributes in a `single line`.

Coloring in both RGB and CMYK

This an external link to the Prawn wiki

There are two other text methods available: `formatted_text` and `formatted_text_box`.

These are useful when the provided text has numerous portions that need to be formatted differently. As you might imply from their names the first should be used for free flowing text just like the `text` method and the last should be used for positioned text just like `text_box`.

The main difference between these methods and the `text` and `text_box` methods is how the text is provided. The `formatted_text` and `formatted_text_box` methods accept an array of hashes. Each hash must provide a `:text` option which is the text string and may provide the following options: `:styles` (an array of symbols), `:size` (the font size), `:character_spacing` (additional space between the characters), `:font` (the name of a registered font), `:color` (the same input accepted by `fill_color` and `stroke_color`), `:link` (an URL to create a link), and `:local` (a link to a local file).

```
formatted_text [ { :text => "Some bold. ",      :styles => [:bold] },
                 { :text => "Some italic. ",    :styles => [:italic] },
                 { :text => "Bold italic. ",    :styles => [:bold, :italic] },
                 { :text => "Bigger text. ",    :size   => 20 },
                 { :text => "More spacing. ",   :character_spacing => 3 },
                 { :text => "Different font. ", :font => "Courier" },
                 { :text => "Some coloring. ",  :color => "990000" },
                 { :text => "Link to the wiki. ",
                   :color => "000099",
                   :link => "https://github.com/prawnpdf/prawn/wiki" },
                 { :text => "Link to a local file. ",
                   :color => "000099",
                   :local => "./local_file.txt" }
               ]

formatted_text_box [ { :text => "Just your regular " },
                     { :text => "text_box ", :font => "Courier" },
                     { :text => "with some additional formatting options " +
                                "added to the mix.",
                       :color => [50, 100, 0, 0],
                       :styles => [:italic] }
                   ], :at => [100, 100], :width => 200, :height => 100
```


The `:callback` option is also available for the formatted text methods.

This option accepts an object (or array of objects) on which two methods will be called if defined: **`render_behind`** and **`render_in_front`**. They are called before and after rendering the text fragment and are passed the fragment as an argument.

This example defines two new callback classes and provide callback objects for the formatted_text

```
class HighlightCallback
  def initialize(options)
    @color    = options[:color]
    @document = options[:document]
  end

  def render_behind(fragment)
    original_color      = @document.fill_color
    @document.fill_color = @color
    @document.fill_rectangle(fragment.top_left,
                             fragment.width,
                             fragment.height)
    @document.fill_color = original_color
  end
end

class ConnectedBorderCallback
  def initialize(options)
    @radius   = options[:radius]
    @document = options[:document]
  end

  def render_in_front(fragment)
    @document.stroke_polygon(fragment.top_left, fragment.top_right,
                             fragment.bottom_right, fragment.bottom_left)

    @document.fill_circle(fragment.top_left,     @radius)
    @document.fill_circle(fragment.top_right,    @radius)
    @document.fill_circle(fragment.bottom_right, @radius)
    @document.fill_circle(fragment.bottom_left,  @radius)
  end
end

highlight = HighlightCallback.new(:color => 'FFFF00', :document => self)
border    = ConnectedBorderCallback.new(:radius => 2.5, :document => self)

formatted_text [ { :text => "hello", :callback => highlight },
                 { :text => "    " },
                 { :text => "world", :callback => border },
                 { :text => "    " },
                 { :text => "hello world", :callback => [highlight, border] }
               ], :size => 20
```

hello world hello world

You have already seen how to set the text color using both inline formatting and the format text methods. There is another way by using the graphics methods `fill_color` and `stroke_color`.

When reading the graphics reference you learned about fill and stroke. If you haven't read it before, read it now before continuing.

Text can be rendered by being filled (the default mode) or just stroked or both filled and stroked. This can be set using the `text_rendering_mode` method or the `:mode` option on the text methods.

```
fill_color   "00ff00"
stroke_color "0000ff"

font_size(48) do
  # normal rendering mode: fill
  text "This text is filled with green."
  move_down 20

  # inline rendering mode: stroke
  text "This text is stroked with blue", mode => :stroke
  move_down 20

  # block rendering mode: fill and stroke
  text_rendering_mode(:fill_stroke) do
    text "This text is filled with green and stroked with blue"
  end
end
```

This text is filled with green.

This text is stroked with blue

This text is filled with green and stroked with blue

We've already seen one way of using text boxes with the `text_box` method. Turns out this method is just a convenience for using the `Prawn::Text::Box` class as it creates a new object and call `render` on it.

Knowing that any extensions we add to `Prawn::Text::Box` will take effect when we use the `text_box` method. To add an exension all we need to do is append the `Prawn::Text::Box.extensions` array with a module.

```
module TriangleBox
  def available_width
    height - 25
  end
end

y_position = cursor - 10
width      = 100
height     = 100

Prawn::Text::Box.extensions << TriangleBox
stroke_rectangle([0, y_position], width, height)
text_box("A" * 100,
         [at => [0, y_position],
         :width => width,
         :height => height)

Prawn::Text::Formatted::Box.extensions << TriangleBox
stroke_rectangle([200, y_position], width, height)
formatted_text_box([{text => "A" * 100, :color => "009900"}],
                   [at => [200, y_position],
                   :width => width,
                   :height => height)

# Once we clear the extensions array
Prawn::Text::Box.extensions.clear
Prawn::Text::Formatted::Box.extensions.clear
```

The PDF format has some built-in font support. If you want to use other fonts in Prawn you need to embed the font file.

Doing this for a single font is extremely simple. Remember the Styling font example? Another use of the `font` method is to provide a font file path and the font will be embedded in the document and set as the current font.

This is reasonable if a font is used only once, but, if a font used several times, providing the path each time it is used becomes cumbersome. The example on the next page shows a better way to deal with fonts which are used several times in a document.

```
# Using a TTF font file
font("#{Prawn::DATADIR}/fonts/DejaVuSans.ttf") do
  text "Written with the DejaVu Sans TTF font."
end
move_down 20

text "Written with the default font."
move_down 20

# Using an DFONT font file
font("#{Prawn::DATADIR}/fonts/Panic Sans.dfont") do
  text "Written with the Panic Sans DFONT font"
end
move_down 20

text "Written with the default font once more."
```

Written with the DejaVu Sans TTF font.

Written with the default font.

Written with the Panic Sans DFONT font

Written with the default font once more.

Registering font families will help you when you want to use a font over and over or if you would like to take advantage of the `:style` option of the text methods and the `b` and `i` tags when using inline formatting.

To register a font family update the `font_families` hash with the font path for each style you want to use.

```
# Registering a single TTF font
font_families.update(
  "DejaVu Sans" => {
    :normal => "#{prawn:DATADIR}/fonts/DejaVuSans.ttf"
  }
)

font("DejaVu Sans") do
  text "Using the DejaVu Sans font providing only its name to the font method"
end
move_down 20

# Registering a DFONT package
font_path = "#{prawn:DATADIR}/fonts/Panic Sans.dfont"
font_families.update(
  "Panic Sans" => {
    :normal      => { :file => font_path, :font => "PanicSans" },
    :italic      => { :file => font_path, :font => "PanicSans Italic" },
    :bold        => { :file => font_path, :font => "PanicSans Bold" },
    :bold_italic => { :file => font_path, :font => "PanicSans BoldItalic" }
  }
)

font "Panic Sans"
text "Also using Panic Sans by providing only its name"
move_down 20

text "Taking <b>advantage</b> of the <i>inline formatting</i>",
     :inline_format => true
move_down 20

[:bold, :bold_italic, :italic, :normal].each do |style|
  text "Using the #{style} style option.",
       :style => style
  move_down 10
end
```

Using the DejaVu Sans font providing only its name to the font method

Also using Panic Sans by providing only its name

Taking **advantage** of the *inline formatting*

Using the bold style option.

Using the bold_italic style option.

Using the italic style option.

Using the normal style option.

Multilingualization isn't much of a problem on Prawn as its default encoding is UTF-8. The only thing you need to worry about is if the font support the glyphs of your language.

```
text "Take this example, a simple Euro sign:"
text "€", size => 32
move_down 20

text "This works, because € is one of the few " +
     "non-ASCII glyphs supported in PDF built-in fonts."

move_down 20

text "For full internationalized text support, we need to use TTF fonts:"
move_down 20

font("#{Prawn::DATADIR}/fonts/DejaVuSans.ttf") do
  text "Θαλον φαγεΐν δύναμαι· τοῦτο οὔ με βλάπτει."
  text "There you go."
end
```

Take this example, a simple Euro sign:

€

This works, because € is one of the few non-ASCII glyphs supported in PDF built-in fonts.

For full internationalized text support, we need to use TTF fonts:

Θαλον φαγεΐν δύναμαι· τοῦτο οὔ με βλάπτει.
There you go.

Line wrapping happens on white space or hyphens. Soft hyphens can be used to indicate where words can be hyphenated. Non-breaking spaces can be used to display space without allowing for a break.

For writing styles that do not make use of spaces, the zero width space serves to mark word boundaries. Zero width spaces are available only with TTF fonts.

```
text "Hard hyphens:\n" +
     "Slip-sliding away, slip sliding awaaaay. You know the " +
     "nearer your destination the more you're slip-sliding away."
move_down 20

shy = Prawn::Text::SHY
text "Soft hyphens:\n" +
     "Slip slid#{shy}ing away, slip slid#{shy}ing away. You know the " +
     "nearer your destinat#{shy}ion the more you're slip slid#{shy}ing away."
move_down 20

nbsp = Prawn::Text::NBSP
text "Non-breaking spaces:\n" +
     "Slip#{nbsp}sliding away, slip#{nbsp}sliding awaaaay. You know the " +
     "nearer your destination the more you're slip#{nbsp}sliding away."
move_down 20

font("#{Prawn::DATADIR}/fonts/gkai00mp.ttf", :size => 16) do
  long_text = "No word boundaries:\n[illegible]"
  text long_text
  move_down 20

  zwsp = Prawn::Text::ZWSP
  long_text = "Invisible word boundaries:\n[illegible]"
  text long_text
end
```

Hard hyphens:
Slip-sliding away, slip sliding awaaaay. You know the nearer your destination the more you're slip-sliding away.

Soft hyphens:
Slip sliding away, slip sliding away. You know the nearer your destination the more you're slip sliding away.

Non-breaking spaces:
Slip sliding away, slip sliding awaaaay. You know the nearer your destination the more you're slip sliding away.

No word boundaries:
更可怕的是，同质化竞争对手可以按照URL中后面这个ID来遍历您的DB中的内容。写个小爬虫把你的页面上的关键信息顺次爬下来也不是什么难事，这样的话，你就非常被动了。更可怕的是，同质化竞争对手可以按照URL中后面这个ID来遍历您的DB中的内容。写个小爬虫把你的页面上的关键信息顺次爬下来也不是什么难事，这样的话，你就非常被动了。

Invisible word boundaries:
更可怕的是，同质化竞争对手可以按照URL中后面这个ID来遍历您的DB中的内容。写个小爬虫把你的页面上的关键信息顺次爬下来也不是什么难事，这样的话，你就非常被动了。更可怕的是，同质化竞争对手可以按照URL中后面这个ID来遍历您的DB中的内容。写个小爬虫把你的页面上的关键信息顺次爬下来也不是什么难事，这样的话，你就非常被动了。

text/right_to_left_text.rb

Prawn can be used with right-to-left text. The direction can be set document-wide, on particular text, or on a text-box. Setting the direction to `:rtl` automatically changes the default alignment to `:right`

You can even override direction on an individual fragment. The one caveat is that two fragments going against the main direction cannot be placed next to each other without appearing in the wrong order.

Writing bidirectional text that combines both left-to-right and right-to-left languages is easy using the `bidi` Ruby Gem and its `render_visual` function. See https://github.com/elad/ruby-bidi for instructions and an example using Prawn.

```ruby
# set the direction document-wide
self.text_direction = :rtl

font("#{Prawn::DATADIR}/fonts/gkai00mp.ttf", :size => 16) do
  long_text =
"写个小爬虫把你的页面上的关键信息顺次爬下来也不是什么难事写个小爬虫把你的页面上的关键信息顺次爬下来也不是什么难事写个小爬虫把你的页面上的关键信息顺次爬下来也不是什么难事写个小爬虫"
  text long_text
  move_down 20

  text "You can override the document direction.", :direction => :ltr
  move_down 20

  formatted_text [ { :text => "更可怕的是，同质化竞争对手可以按照" },
                   { :text => "URL", :direction => :ltr },
                   { :text => "中后面这个" },
                   { :text => "ID", :direction => :ltr },
                   { :text => "来遍历您的" },
                   { :text => "DB", :direction => :ltr },
                   { :text =>
"中的内容，写个小爬虫把你的页面上的关键信息顺次爬下来也不是什么难事，这样的话，你就非常被动了。"
  }]
  move_down 20

  formatted_text [ { :text => "更可怕的是，同质化竞争对手可以按照" },
                   { :text => "this",  :direction => :ltr },
                   { :text => "won't", :direction => :ltr, :size => 24 },
                   { :text => "work",  :direction => :ltr },
                   { :text =>
"中的内容，写个小爬虫把你的页面上的关键信息顺次爬下来也不是什么难事" }]
end
```

虫爬小个写事难么什是不也来下爬次顺息信键关的上面页的你把虫爬小个写
面页的你把虫爬小个写事难么什是不也来下爬次顺息信键关的上面页的你把
小个写事难么什是不也来下爬次顺息信键关的上

You can override the document direction.

内的中DB的您历遍来ID个这面后中URL照按以可手对争竞化质同，是的怕可更
样这，事难么什是不也来下爬次顺息信键关的上面页的你把虫爬小个写，容

。了的做需要就你，话的

例小个写，容内的中workwon't this照能以可手对争竞化质同，是的信可更
事难么什是不也来下做次哪息信键关的上面页的你把来

Prawn enables the declaration of fallback fonts for those glyphs that may not be present in the desired font. Use the `:fallback_fonts` option with any of the text or text box methods, or set fallback_fonts document-wide.

```
file = "#{Prawn::DATADIR}/fonts/gkai00mp.ttf"
font_families["kai"] = {
  :normal => { :file => file, :font => "kai" }
}

file = "#{Prawn::DATADIR}/fonts/Ponic-Sans.dfont"
font_families["Ponic Sans"] = {
  :normal => { :file => file, :font => "Fairchere" }
}

font("Ponic Sans") do
  text("When fallback fonts are included, each glyph will be rendered " +
       "using the first font that includes the glyph, starting with the " +
       "current font and then moving through the fallback fonts from left " +
       "to right." +
       "\n\n" +
       "hello ƒ 你好\n再见 ƒ goodbye",
       :fallback_fonts => ["Times Roman", "kai"])
end
move_down 20

formatted_text([ { :text => "Fallback fonts can even override" },
                 { :text => "fragment fonts (你好)", :font => "Times Roman" }
               ],
               :fallback_fonts => ["Times Roman", "kai"])
```

When fallback fonts are included, each glyph will be rendered using the first font that includes the glyph, starting with the current font and then moving through the fallback fonts from left to right.

hello ƒ 你好
再见 ƒ goodbye

Fallback fonts can even override fragment fonts (你好)

Prints a list of all of the glyphs that can be rendered by Adobe's built in fonts, along with their character widths and WinAnsi codes. Be sure to pass these glyphs as UTF-8, and Prawn will transcode them for you.

```
FONT_SIZE = 9.5

x = 0
y = bounds.top

fields = [[20, :right], [6, :left], [12, :center], [30, :right], [6, :left],
          [6, :left]]

font "Helvetica", :size => FONT_SIZE

move_down 20
text "(See next page for WinAnsi table)", :align => :center
start_new_page

Prawn::Encoding::WinAnsi::CODE2NAME.each_with_index do |name, index|
  next if name == ".notdef"
  y -= FONT_SIZE

  if y < FONT_SIZE
    y = bounds.top - FONT_SIZE
    x += 170
  end

  code = "%i." % index
  char = index.chr

  width = 1000 * width_of(char, :size => FONT_SIZE) / FONT_SIZE
  size  = "%3" % width

  data = [code, nil, char, size, nil, name]
  dx   = x
  fields.zip(data).each do |total_width, align|, field|
    if field
      width = width_of(field, :size => FONT_SIZE)

      case align
      when :left   then offset = 0
      when :right  then offset = total_width - width
      when :center then offset = (total_width - width) / 2
      end

      text_box(field.force_encoding("windows-1252").encode("UTF-8"),
               :at => [dx + offset, y])
    end

    dx += total_width
  end
end
```

(See next page for WinAnsi table)

32		278	space
33	!	278	exclam
34	"	355	quotedbl
35	#	556	numbersign
36	$	556	dollar
37	%	889	percent
38	&	667	ampersand
39	'	191	quotesingle
40	(	333	parenleft
41	)	333	parenright
42	*	389	asterisk
43	+	584	plus
44	,	278	comma
45	-	333	hyphen
46	.	278	period
47	/	278	slash
48	0	556	zero
49	1	556	one
50	2	556	two
51	3	556	three
52	4	556	four
53	5	556	five
54	6	556	six
55	7	556	seven
56	8	556	eight
57	9	556	nine
58	:	278	colon
59	;	278	semicolon
60	<	584	less
61	=	584	equal
62	>	584	greater
63	?	556	question
64	@	1015	at
65	A	667	A
66	B	667	B
67	C	722	C
68	D	722	D
69	E	667	E
70	F	611	F
71	G	778	G
72	H	722	H
73	I	278	I
74	J	500	J
75	K	667	K
76	L	556	L
77	M	833	M
78	N	722	N
79	O	778	O
80	P	667	P
81	Q	778	Q
82	R	722	R
83	S	667	S
84	T	611	T
85	U	722	U
86	V	667	V
87	W	944	W
88	X	667	X
89	Y	667	Y
90	Z	611	Z
91	[	278	bracketleft
92	\	278	backslash
93	]	278	bracketright
94	^	469	asciicircum
95	_	556	underscore
96	`	333	grave
97	a	556	a
98	b	556	b
99	c	500	c
100	d	556	d
101	e	556	e
102	f	278	f
103	g	556	g
104	h	556	h
105	i	222	i
106	j	222	j
107	k	500	k
108	l	222	l
109	m	833	m
110	n	556	n
111	o	556	o
112	p	556	p
113	q	556	q
114	r	333	r
115	s	500	s
116	t	278	t
117	u	556	u
118	v	500	v
119	w	722	w
120	x	500	x
121	y	500	y
122	z	500	z
123	{	334	braceleft
124	|	259	bar
125	}	334	braceright
126	~	584	asciitilde
128	€	556	Euro
130	‚	222	quotesinglbase
131	ƒ	556	florin
132	„	333	quotedblbase
133	…	1000	ellipsis
134	†	556	dagger
135	‡	556	daggerdbl
136	ˆ	333	circumflex
137	‰	1000	perthousand
138	Š	667	Scaron
139	‹	333	guilsinglleft
140	Œ	1000	OE
142	Ž	611	Zcaron
145	'	222	quoteleft
146	'	222	quoteright
147	"	333	quotedblleft
148	"	333	quotedblright
149	•	349	bullet
150	–	556	endash
151	—	1000	emdash
152	˜	333	tilde
153	™	1000	trademark
154	š	500	scaron
155	›	333	guilsinglright
156	œ	944	oe
158	ž	500	zcaron
159	Ÿ	500	ydieresis
160		278	space
161	¡	333	exclamdown
162	¢	556	cent
163	£	556	sterling
164	¤	556	currency
165	¥	556	yen
166	¦	259	brokenbar
167	§	556	section
168	¨	333	dieresis
169	©	737	copyright
170	ª	370	ordfeminine
171	«	556	guillemotleft
172	¬	584	logicalnot
173	-	333	hyphen
174	®	737	registered
175	¯	333	macron
176	°	400	degree
177	±	584	plusminus
178	²	333	twosuperior
179	³	333	threesuperior
180	´	333	acute
181	µ	556	mu
182	¶	537	paragraph
183	·	278	periodcentered
184	¸	333	cedilla
185	¹	333	onesuperior
186	º	365	ordmasculine
187	»	556	guillemotright
188	¼	834	onequarter
189	½	834	onehalf
190	¾	834	threequarters
191	¿	611	questiondown
192	À	667	Agrave
193	Á	667	Aacute
194	Â	667	Acircumflex
195	Ã	667	Atilde
196	Ä	667	Adieresis
197	Å	667	Aring
198	Æ	1000	AE
199	Ç	722	Ccedilla
200	È	667	Egrave
201	É	667	Eacute
202	Ê	667	Ecircumflex
203	Ë	667	Edieresis
204	Ì	278	Igrave
205	Í	278	Iacute
206	Î	278	Icircumflex
207	Ï	278	Idieresis
208	Ð	722	Eth
209	Ñ	722	Ntilde
210	Ò	778	Ograve
211	Ó	778	Oacute
212	Ô	778	Ocircumflex
213	Õ	778	Otilde
214	Ö	778	Odieresis
215	×	584	multiply
216	Ø	778	Oslash
217	Ù	722	Ugrave
218	Ú	722	Uacute
219	Û	722	Ucircumflex
220	Ü	722	Udieresis
221	Ý	667	Yacute
222	Þ	667	Thorn
223	ß	611	germandbls
224	à	556	agrave
225	á	556	aacute
226	â	556	acircumflex
227	ã	556	atilde
228	ä	556	adieresis
229	å	556	aring
230	æ	889	ae
231	ç	500	ccedilla
232	è	556	egrave
233	é	556	eacute
234	ê	556	ecircumflex
235	ë	556	edieresis
236	ì	278	igrave
237	í	278	iacute
238	î	278	icircumflex
239	ï	278	idieresis
240	ð	556	eth
241	ñ	556	ntilde
242	ò	556	ograve
243	ó	556	oacute
244	ô	556	ocircumflex
245	õ	556	otilde
246	ö	556	odieresis
247	÷	584	divide
248	ø	611	oslash
249	ù	556	ugrave
250	ú	556	uacute
251	û	556	ucircumflex
252	ü	556	udieresis
253	ý	500	yacute
254	þ	556	thorn
255	ÿ	500	ydieresis

Bounding box

Bounding boxes are the basic containers for structuring the content flow. Even being low level building blocks sometimes their simplicity is very welcome.

The examples show:

- How to create bounding boxes with specific dimensions
- How to inspect the current bounding box for its coordinates
- Stretchy bounding boxes
- Nested bounding boxes
- Indent blocks

bounding_box/creation.rb

If you've read the basic concepts examples you probably know that the origin of a page is on the bottom left corner and that the content flows from top to bottom.

You also know that a Bounding Box is a structure for helping the content flow.

A bounding box can be created with the `bounding_box` method. Just provide the top left corner, a required `:width` option and an optional `:height`.

```
bounding_box[200, cursor - 100], :width => 200, :height => 100) do
  text "Just your regular bounding box"

  transparent(0.5) { stroke_bounds }
end
```

The `bounds` method returns the current bounding box. This is useful because the `Prawn::BoundingBox` exposes some nice boundary helpers.

`top`, `bottom`, `left` and `right` methods return the bounding box boundaries relative to its translated origin. `top_left`, `top_right`, `bottom_left` and `bottom_right` return those boundaries pairs inside arrays.

All these methods have an "absolute" version like `absolute_right`. The absolute version returns the same boundary relative to the page absolute coordinates.

The following snippet shows the boundaries for the margin box side by side with the boundaries for a custom bounding box.

```
def print_coordinates
  text "top: #{bounds.top}"
  text "bottom: #{bounds.bottom}"
  text "left: #{bounds.left}"
  text "right: #{bounds.right}"

  move_down 10

  text "absolute top: #{sprintf "%.2f", bounds.absolute_top}"
  text "absolute bottom: #{sprintf "%.2f", bounds.absolute_bottom}"
  text "absolute left: #{sprintf "%.2f", bounds.absolute_left}"
  text "absolute right: #{sprintf "%.2f", bounds.absolute_right}"
end

text "Margin box bounds:"
move_down 5
print_coordinates

bounding_box([250, cursor - 140], :width => 200, :height => 150) do
  text "This bounding box bounds:"
  move_down 5
  print_coordinates
  transparent(0.5) { stroke_bounds }
end
```

Margin box bounds:

top: 864.0
bottom: 0
left: 0
right: 540.0

absolute top: 900.00
absolute bottom: 36.00
absolute left: 36.00
absolute right: 576.00

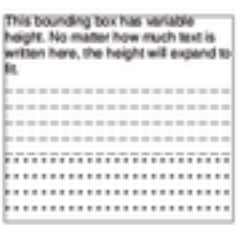
This bounding box bounds:

top: 150
bottom: 0
left: 0
right: 200

absolute top: 263.17
absolute bottom: 113.17
absolute left: 286.00
absolute right: 486.00

Bounding Boxes accept an optional `:height` parameter. Unless it is provided the bounding box will be stretchy. It will expand the height to fit all content generated inside it.

```
y_position = cursor
bounding_box[0, y_position], :width => 200, :height => 100) do
  text "This bounding box has a height of 100. If this text gets too large " +
       "it will flow to the next page."

  transparent(0.5) { stroke_bounds }
end

bounding_box[300, y_position], :width => 200) do
  text "This bounding box has variable height. No matter how much text is " +
       "written here, the height will expand to fit."

  text " _" * 100

  text " -" * 100

  transparent(0.5) { stroke_bounds }
end
```



Normally when we provide the top left corner of a bounding box we express the coordinates relative to the *margin* box. This is not the case when we have nested bounding boxes. Once nested the inner bounding box coordinates are relative to the outer bounding box.

This example shows some nested bounding boxes with fixed and stretchy heights. Note how the `cursor` method returns coordinates relative to the current bounding box.

```
def box_content(string)
  text string
  transparent(0.5) { stroke_bounds }
end

gap = 20
bounding_box([50, cursor], width: 400, height: 200) do
  box_content("Fixed height")

  bounding_box([gap, cursor - gap], width: 300) do
    text "Stretchy height"

    bounding_box([gap, bounds.top - gap], width: 100) do
      text "Stretchy height"
      transparent(0.5) { dash(1); stroke_bounds; undash }
    end

    bounding_box([gap * 7, bounds.top - gap], width: 100, height: 50) do
      box_content("Fixed height")
    end

    transparent(0.5) { dash(1); stroke_bounds; undash }
  end

  bounding_box([gap, cursor - gap], width: 300, height: 50) do
    box_content("Fixed height")
  end
end
```

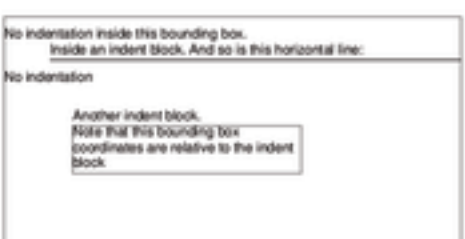
bounding_box/indentation.rb

Sometimes you just need to indent a portion of the contents of a bounding box, and using a nested bounding box is pure overkill. The `indent` method is what you might need.

Just provide a number for it to indent all content generated inside the block.

```
text "No indentation on the margin box."
indent(20) do
  text "Some indentation inside an indent block."
end
move_down 20

bounding_box([50, cursor], :width => 400, :height => cursor) do
  transparent(0.5) { stroke_bounds }

  move_down 10
  text "No indentation inside this bounding box."
  indent(40) do
    text "Inside an indent block. And so is this horizontal line:"

    stroke_horizontal_rule
  end
  move_down 10
  text "No indentation"

  move_down 20
  indent(60) do
    text "Another indent block."

    bounding_box([0, cursor], :width => 200) do
      text "Note that this bounding box coordinates are relative to the " +
           "indent block"

      transparent(0.5) { stroke_bounds }
    end
  end
end
```

No indentation on the margin box.
Some indentation inside an indent block.



The origin example already mentions that a new document already comes with a margin box whose bottom left corner is used as the origin for calculating coordinates.

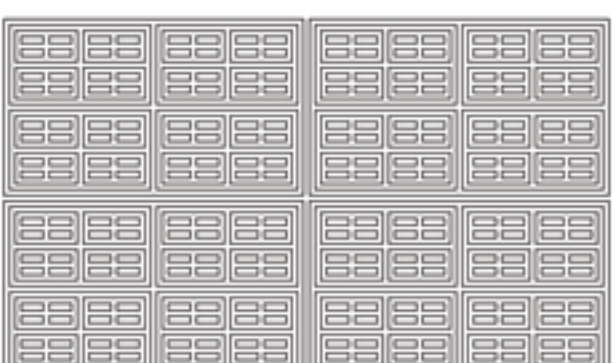
What has not been told is that there is one helper for "bypassing" the margin box: `canvas`. This method is a shortcut for creating a bounding box mapped to the absolute coordinates and evaluating the code inside it.

The following snippet draws a circle on each of the four absolute corners.

```
canvas do
  fill_circle [bounds.left, bounds.top],     10
  fill_circle [bounds.right, bounds.top],    10
  fill_circle [bounds.right, bounds.bottom], 10
  fill_circle [0, 0],                        10
end
```

This example is mostly just for fun, and shows how nested bounding boxes can simplify calculations. See the "Bounding Box" section of the manual for more basic uses.

```
def combine(a1, a2)
  output = []
  a1.each do |i1|
    a2.each do |i2|
      output << [i1, i2]
    end
  end
  output
end

def recurse_bounding_box(max_depth = 4, depth = 1)
  width = (bounds.width - 15) / 2
  height = (bounds.height - 15) / 2
  left_top_corners = combine([5, bounds.right - width - 5],
                             [bounds.top - 5, height + 5])
  left_top_corners.each do |x|
    bounding_box(x, width: width, height: height) do
      stroke_bounds
      recurse_bounding_box(max_depth, depth + 1) if depth < max_depth
    end
  end
end

# Set up a base from the dashed line to the bottom of the page
bounding_box([0, cursor], width: bounds.width, height: cursor) do
  recurse_bounding_box
end
```



Layout

Prawn has support for two-dimensional grid based layouts out of the box.

The examples show:

- How to define the document grid
- How to configure the grid rows and columns gutters
- How to create boxes according to the grid

layout/simple_grid.rb

The document grid on Prawn is just a table-like structure with a defined number of rows and columns. There are some helpers to create boxes of content based on the grid coordinates.

`define_grid` accepts the following options which are pretty much self-explanatory: `:rows`, `:columns`, `:gutter`, `:row_gutter`, `:column_gutter`

```
# The grid only need to be defined once, but since all the examples should be
# able to run alone we are repeating it on every example
define_grid(columns => 5, :rows => 8, :gutter => 10)
text "We defined the grid, roll over to the next page to see its outline"

start_new_page
grid.show_all
```

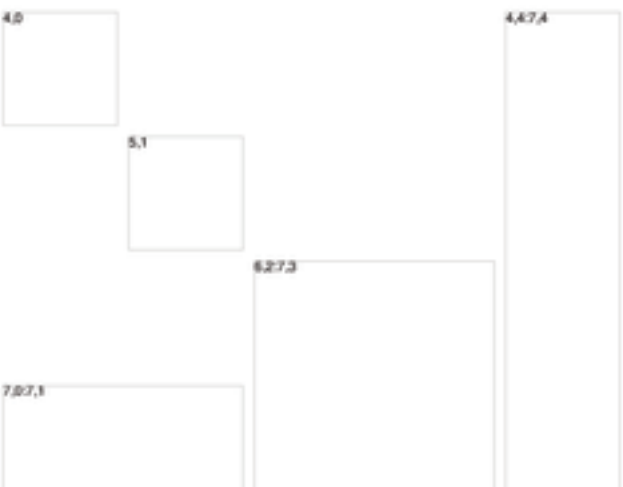
We defined the grid, roll over to the next page to see its outline


0,0
0,1
0,2
0,3
0,4
1,0
1,1
1,2
1,3
1,4
2,0
2,1
2,2
2,3
2,4
3,0
3,1
3,2
3,3
3,4
4,0
4,1
4,2
4,3
4,4
5,0
5,1
5,2
5,3
5,4
6,0
6,1
6,2
6,3
6,4
7,0
7,1
7,2
7,3
7,4

After defined the grid is there but nothing happens. To start taking effect we need to use the grid boxes.

`grid` has three different return values based on the arguments received. With no arguments it will return the grid itself. With integers it will return the grid box at those indices. With two arrays it will return a multi-box spanning the region of the two grid boxes at the arrays indices.

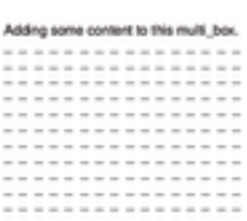
```
# The grid only need to be defined once, but since all the examples should be
# able to run alone we are repeating it on every example
define_grid(columns => 5, rows => 8, gutter => 10)

grid(4, 0).show
grid(5, 1).show

grid(6, 2), [7, 3]).show

grid(4, 4), [7, 4]).show
grid(7, 0), [7, 1]).show
```



Now that we know how to access the boxes we might as well add some content to them.

This can be done by taping into the bounding box for a given grid box or multi-box with the **bounding_box** method.

```
# the grid only need to be defined once, but since all the examples should be
# able to run alone we are repeating it on every example
define_grid(columns => 5, rows => 8, gutter => 10)

grid(3, 0, 1, 1).bounding_box do
  text "Adding some content to this multi_box.\n" * " _ " * 200
end

grid(4, 3).bounding_box do
  text "Just a little snippet here.\n" * " _ " * 10
end
```



Prawn::Table

As of Prawn 1.2.0, Prawn::Table has been extracted into its own semi-officially supported gem.

Please see https://github.com/prawnpdf/prawn-table for more details.

Images

Embedding images on PDF documents is fairly easy. Prawn supports both JPG and PNG images.

The examples show:

- How to add an image to a page
- How place the image on a specific position
- How to configure the image dimensions by setting the width and height or by scaling it

images/plain_image.rb

To embed images onto your PDF file use the `image` method. It accepts the file path of the image to be loaded and some optional arguments.

If only the image path is provided the image will be rendered starting on the cursor position. No manipulation is done with the image even if it doesn't fit entirely on the page like the following snippet.

```
text "The image will go right below this line of text."
image "#{Prawn::DATADIR}/images/pigs.jpg"
```

The image will go right below this line of text.

images/absolute_position.rb

One of the options that the `image` method accepts is `:at`. If you've read some of the graphics examples you are probably already familiar with it. Just provide it the upper-left corner where you want the image placed.

While sometimes useful this option won't be practical. Notice that the cursor won't be moved after the image is rendered and there is nothing forbidding the text to overlap with the image.

```
y_position = cursor
text "The image won't go below this line of text."

image "#{Prawn::DATADIR}/images/fractal.jpg", at: [200, y_position]

text "And this line of text will go just below the previous one."
```

The image won't go below this line of
And this line of text will go just below the previous one.

The image may be positioned relatively to the current bounding box. The horizontal position may be set with the `:position` option.

It may be `:left`, `:center`, `:right`, or a number representing an x-offset from the left boundary.

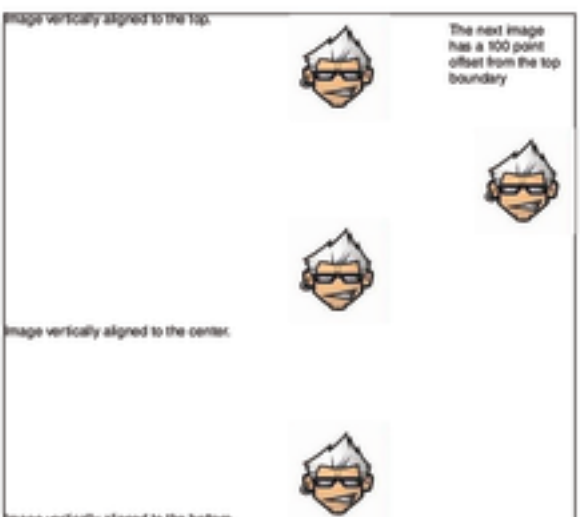
```
bounding_box([50, cursor], :width => 400, :height => 450) do
  stroke_bounds

  [:left, :center, :right].each do |position|
    text "Image aligned to the #{position}."
    image "#{Prawn::DATADIR}/images/mtf.jpg", :position => position
  end

  text "The next image has a 50 point offset from the left boundary"
  image "#{Prawn::DATADIR}/images/mtf.jpg", :position => 50
end
```

images/vertical.rb

To set the vertical position of an image use the `:valign` option.

It may be `:top`, `:center`, `:bottom` or a number representing the y-offset from the top boundary.

```
bounding_box[0, cursor], width: 500, height: 450 do
  stroke_bounds

  [:top, :center, :bottom].each do |position|
    text "Image vertically aligned to the #{position}.", valign: position
    image "#{Prawn::DATADIR}/images/stef.jpg", position:  :left,
                                               vposition: position
  end

  text_box "The next image has a 100 point offset from the top boundary",
           :at => [bounds.width - 150, bounds.top - 10], :width => 130
  image "#{Prawn::DATADIR}/images/stef.jpg", position:  :right,
                                             vposition: 100
end
```



images/width_and_height.rb

The image size can be set with the `:width` and `:height` options.

If only one of those is provided, the image will be scaled proportionally. When both are provided, the image will be stretched to fit the dimensions without maintaining the aspect ratio.

```
text "Scale by setting only the width"
image "#{Prawn::DATADIR}/images/pigs.jpg", :width => 150
move_down 20

text "Scale by setting only the height"
image "#{Prawn::DATADIR}/images/pigs.jpg", :height => 100
move_down 20

text "Stretch to fit the width and height provided"
image "#{Prawn::DATADIR}/images/pigs.jpg", :width => 500, :height => 150
```

Scale by setting only the width

Scale by setting only the height

Stretch to fit the width and height provided

images/scale.rb

To scale an image use the `:scale` option.

It scales the image proportionally given the provided value.

```
text  "Normal size"
image "#{Prawn::DATADIR}/images/vtxf.jpg"
move_down 20

text  "Scaled to 50%"
image "#{Prawn::DATADIR}/images/vtxf.jpg", :scale => 0.5
move_down 20

text  "Scaled to 200%"
image "#{Prawn::DATADIR}/images/vtxf.jpg", :scale => 2
```

Normal size

Scaled to 50%

Scaled to 200%

`:fit` option is useful when you want the image to have the maximum size within a container preserving the aspect ratio without overlapping.

Just provide the container width and height pair.

```ruby
size = 300

text "Using the fit option"
bounding_box([0, cursor], :width => size, :height => size) do
  image "#{Prawn::DATADIR}/images/pigs.jpg", :fit => [size, size]
  stroke_bounds
end
```

Using the fit option

Document and page options

So far we've already seen how to create new documents and start new pages. This chapter expands on the previous examples by showing other options avialable. Some of the options are only available when creating new documents.

The examples show:

- How to configure page size
- How to configure page margins
- How to use a background image
- How to add metadata to the generated PDF

Prawn comes with support for most of the common page sizes so you'll only need to provide specific values if your intended format is not supported. To see a list with all supported sizes take a look at PDF::Core::PageGeometry

To define the size use :**page_size** when creating new documents and :**size** when starting new pages. The default page size for new documents is LETTER (612.00 x 792.00).

You may also define the orientation of the page to be either *portrait* (default) or landscape. Use :**page_layout** when creating new documents and :**layout** when starting new pages.

```
Prawn::Document.generate("page_size.pdf",
                         :page_size   => "HALFLETTER",
                         :page_layout => :landscape
) do
  text "HALFLETTER landscape page."

  custom_size = [275, 324]

  ["A4", "LEGALSIZE", "LT", custom_size ].each do |size|
    start_new_page(:size => size, :layout => :portrait)
    text "#{size} portrait page."

    start_new_page(:size => size, :layout => :landscape)
    text "#{size} landscape page."
  end
end
```

This code snippet was not evaluated inline. You may see its output by running the example file located here:
http://github.com/prawnpdf/prawn/tree/master/manual/document_and_page_options/page_size.rb

The default margin for pages is 0.5 inch but you can change that with the `:margin` option or if you'd like to have different margins you can use the `:left_margin`, `:right_margin`, `:top_margin`, `:bottom_margin` options.

These options are available both for starting new pages and creating new documents.

```
Prawn::Document.generate("page_margins.pdf",
                         :margin => 100
) do
  text "100 pts margins."
  stroke_bounds

  start_new_page(:left_margin => 300)
  text "300 pts margin on the left."
  stroke_bounds

  start_new_page(:top_margin => 300)
  text "300 pts margin both on the top and on the left. Notice that whenever " +
       "you set an option for a new page it will remain the default for the " +
       "following pages."
  stroke_bounds

  start_new_page(:margin => 50)
  text "50 pts margins. Using the margin option will reset previous specific " +
       "calls to left, right, top and bottom margins."
  stroke_bounds

  start_new_page(:margin => [50, 100, 150, 200])
  text "Order is also the shorthand CSS like syntax used here."
  stroke_bounds
end
```

This code snippet was not evaluated inline. You may see its output by running the example file located here:
http://github.com/prawnpdf/prawn/tree/master/manual/document_and_page_options/page_margins.rb

document_and_page_options/background.rb

Pass an image path to the :background option and it will be used as the background for all pages. This option can only be used on document creation.

```
img = "#{Prawn::DATADIR}/images/letterhead.jpg"

Prawn::Document.generate("background.pdf",
                         :background => img,
                         :margin => 100
) do
  text "My report caption", :size => 16, :align => :right

  move_down font.height * 2

  text "Here is my text explaining this report. " * 20,
       :size => 12, :align => :left, :leading => 2

  move_down font.height

  text "I'm using a soft background. " * 40,
       :size => 12, :align => :left, :leading => 2
end
```

This code snippet was not evaluated inline. You may see its output by running the example file located here:
http://github.com/prawnpdf/prawn/tree/master/manual/document_and_page_options/background.rb

document_and_page_options/metadata.rb

To set the document metadata just pass a hash to the :info option when creating new documents. The keys in the example below are arbitrary, so you may add whatever keys you want

```
info = {
  :Title        => "My title",
  :Author       => "John Doe",
  :Subject      => "My Subject",
  :Keywords     => "test metadata ruby pdf dry",
  :Creator      => "Adobe Soft App",
  :Producer     => "Prawn",
  :CreationDate => Time.now
}

Prawn::Document.generate("metadata.pdf", :info => info) do
  text "This is a test of setting metadata properties via the info option."
  text "While the keys are arbitrary, the above example sets common attributes."
end
```

This code snippet was not evaluated inline. You may see its output by running the example file located here:

http://github.com/prawnpdf/prawn/tree/master/manual/document_and_page_options/metadata.rb

(Optional; PDF 1.6) The page scaling option to be selected when a print dialog is displayed for this document. Valid values are **None**, which indicates that the print dialog should reflect no page scaling, and **AppDefault**, which indicates that applications should use the current print scaling. If this entry has an unrecognized value, applications should use the current print scaling. Default value: **AppDefault**.

Note: If the print dialog is suppressed and its parameters are provided directly by the application, the value of this entry should still be used.

```
Prawn::Document.generate("print_scaling.pdf",
                         :page_layout => :landscape,
                         :print_scaling => :none
) do
  text
"When you print this document, the scale to fit in print preview should be disab
led by default."
end
```

This code snippet was not evaluated inline. You may see its output by running the example file located here:
http://github.com/prawnpdf/prawn/tree/master/manual/document_and_page_options/print_scaling.rb

Outline

The outline of a PDF document is the table of contents tab you see to the right or left of your PDF viewer.

The examples include:

- How to define sections and pages
- How to insert sections and/or pages to a previously defined outline structure

outline/sections_and_pages.rb

The document outline tree is the set of links used to navigate through the various document sections and pages.

To define the document outline we first use the `outline` method to lazily instantiate an outline object. Then we use the `define` method with a block to start the outline tree.

The basic methods for creating outline nodes are `section` and `page`. The only difference between the two is that `page` doesn't accept a block and will only create leaf nodes while `section` accepts a block to create nested nodes.

`section` accepts the title of the section and two options: `:destination` - a page number to link and `:closed` - a boolean value that defines if the nested outline nodes are shown when the document is open (defaults to true).

`page` is very similar to section. It requires a `:title` option to be set and accepts a `:destination`.

`section` and `page` may also be used without the `define` method but they will need to instantiate the `outline` object every time.

```
# First we create 10 pages just to have something to link to
(1..10).each do |index|
  text "Page #{index}"
  start_new_page
end

outline.define do
  section("Section 1", :destination => 1) do
    page :title => "Page 2", :destination => 2
    page :title => "Page 3", :destination => 3
  end

  section("Section 2", :destination => 4) do
    page :title => "Page 5", :destination => 5

    section("Subsection 2.1", :destination => 6, :closed => true) do
      page :title => "Page 7", :destination => 7
    end
  end
end

# Outside of the define block
outline.section("Section 3", :destination => 8) do
  outline.page :title => "Page 9", :destination => 9
end

outline.page :title => "Page 10", :destination => 10

# Section and Pages without links. While a section without a link may be
# useful to group some pages, a page without a link is useless
outline.update do # update is an alias to define
  section("Section without link") do
    page :title => "Page without link"
  end
end
```

This code snippet was not evaluated inline. You may see its output by running the example file located here:
http://github.com/prawnpdf/prawn/tree/master/manual/outline/sections_and_pages.rb

We have already seen how to define an outline tree sequentially.

If you'd like to add nodes to the middle of an outline tree the **add_subsection_to** may help you.

It allows you to insert sections to the outline tree at any point. Just provide the **title** of the parent section, the **position** you want the new subsection to be inserted (**:first** or **:last** (defaults to **:last**)) and a block to declare the subsection.

The **add_subsection_to** block doesn't necessarily create new sections, it may also create new pages.

If the parent title provided is the title of a page. The page will be converted into a section to receive the subsection created.

```
# First we create 10 pages and some default outline
(1..10).each do |index|
  text "Page #{index}"
  start_new_page
end

outline.define do
  section("Section 1", :destination => 11 do
    page :title => "Page 2", :destination => 2
    page :title => "Page 3", :destination => 3
  end
end

# Now we will start adding nodes to the previous outline
outline.add_subsection_to("Section 1", :first) do
  outline.section("Added later - first position") do
    outline.page :title => "Page 4", :destination => 4
    outline.page :title => "Page 5", :destination => 5
  end
end

outline.add_subsection_to("Section 1") do
  outline.page :title => "Added later - last position",
               :destination => 6
end

outline.add_subsection_to("Added later - first position") do
  outline.page :title => "Another page added later",
               :destination => 7
end

# The title provided is for a page which will be converted into a section
outline.add_subsection_to("Page 1") do
  outline.page :title => "Last page added",
               :destination => 8
end
```

This code snippet was not evaluated inline. You may see its output by running the example file located here: http://github.com/prawnpdf/prawn/tree/master/manual/outline/add_subsection_to.rb

outline/insert_section_after.rb

Another way to insert nodes into an existing outline is the `insert_section_after` method.

It accepts the title of the node that the new section will go after and a block declaring the new section.

As is the case with `add_subsection_to` the section added doesn't need to be a section, it may be just a page.

```
# First we create 10 pages and some default outline
(1..10).each do |index|
  text "Page #{index}"
  start_new_page
end

outline.define do
  section "Section 1", :destination => 11 do
    page :title => "Page 1", :destination => 2
    page :title => "Page 2", :destination => 3
  end
end

# Now we will start adding nodes to the previous outline
outline.insert_section_after("Page 2") do
  outline.section("Section after Page 2") do
    outline.page :title => "Page 4", :destination => 4
  end
end

outline.insert_section_after("Section 1") do
  outline.section("Section after Section 1") do
    outline.page :title => "Page 5", :destination => 5
  end
end

# Adding just a page
outline.insert_section_after("Page 3") do
  outline.page :title => "Page after Page 3", :destination => 6
end
```

This code snippet was not evaluated inline. You may see its output by running the example file located here:
http://github.com/prawnpdf/prawn/tree/master/manual/outline/insert_section_after.rb

Repeatable content

Prawn offers two ways to handle repeatable content blocks. Repeater is useful for content that gets repeated at well defined intervals while Stamp is more appropriate if you need better control of when to repeat it.

There is also one very specific helper for numbering pages.

The examples show:

- How to repeat content on several pages with a single invocation
- How to create a new Stamp
- How to "stamp" the content block on the page
- How to number the document pages with one simple call

The **repeat** method is quite versatile when it comes to define the intervals at which the content block should repeat.

The interval may be a symbol (`:all`, `:odd`, `:even`), an array listing the pages, a range or a **Proc** that receives the page number as an argument and should return true if the content is to be repeated on the given page.

You may also pass an option `:dynamic` to reevaluate the code block on every call which is useful when the content changes based on the page number.

It is also important to say that no matter where you define the repeater it will be applied to all matching pages.

```
repeat(:all) do
  draw_text "All pages", :at => bounds.top_left
end

repeat(:odd) do
  draw_text "Only odd pages", :at => [0, 0]
end

repeat(:even) do
  draw_text "Only even pages", :at => [0, 0]
end

repeat([1, 3, 7]) do
  draw_text "Only on pages 1, 3 and 7", :at => [100, 0]
end

repeat(2..4) do
  draw_text "From the 2nd to the 4th page", :at => [200, 0]
end

repeat(lambda { |pg| pg % 3 == 0 }) do
  draw_text "Every third page", :at => [250, 20]
end

repeat(:all, :dynamic => true) do
  draw_text page_number, :at => [500, 0]
end

10.times do
  start_new_page
  draw_text "A wonderful page", :at => [400, 400]
end
```

This code snippet was not evaluated inline. You may see its output by running the example file located here:
http://github.com/prawnpdf/prawn/tree/master/manual/repeatable_content/repeater.rb

Stamps should be used when you have content that will be included multiple times in a document. Its advantages over creating the content anew each time are:

1. Faster document creation

2. Smaller final document

3. Faster display on subsequent displays of the repeated element because the viewer application can cache the rendered results

The **create_stamp** method does just what it says. Pass it a block with the content that should be generated and the stamp will be created.

There are two methods to render the stamp on a page **stamp** and **stamp_at**. The first will render the stamp as is while the second accepts a point to serve as an offset to the stamp content.

```
create_stamp("approved") do
  rotate(20, :origin => [-5, -5]) do
    stroke_color "FF3333"
    stroke_ellipse [0, 0], 2*6, 15
    stroke_color "000000"

    fill_color "FF3333"
    font("Times-Roman") do
      draw_text "Approved", :at => [-21, -3]
    end
    fill_color "000000"
  end
end

stamp "approved"

stamp_at "approved", [200, 200]
```

repeatable_content/page_numbering.rb

The `number_pages` method is a simple way to number the pages of your document. It should be called towards the end of the document since pages created after the call won't be numbered.

It accepts a string and a hash of options:

`start_count_at` is the value from which to start numbering pages

`total_pages` If provided, will replace `total` with the value given. Useful for overriding the total number of pages when using the start_count_at option.

`page_filter`, which is one of: `:all`, `:odd`, `:even`, an array, a range, or a Proc that receives the page number as an argument and should return true if the page number should be printed on that page.

`color` which accepts the same values as `fill_color`

As well as any option accepted by `text_box`

```
text "This is the first page!"

10.times do
  start_new_page
  text "Here comes yet another page."
end

string = "page <page> of <total>"
# Green page numbers 1 to 7
options = { :at => [bounds.right - 150, 0],
            :width => 150,
            :align => :right,
            :page_filter => (1..7),
            :start_count_at => 1,
            :color => "007700" }
number_pages string, options

# Grey page numbers from 8 on up
options[:page_filter]    = lambda{ |pg| pg > 7 }
options[:start_count_at] = 8
options[:color]          = "333333"
number_pages string, options

start_new_page
text "Psst.  This page isn't numbered and doesn't count towards the total."
```

This code snippet was not evaluated inline. You may see its output by running the example file located here: http://github.com/prawnpdf/prawn/tree/master/manual/repeatable_content/page_numbering.rb

repeatable_content/alternate_page_numbering.rb

Below is the code to generate page numbers that alternate being rendered on the right and left side of the page. The first page will have a "1" in the bottom *right* corner. The second page will have a "2" in the bottom left corner of the page. The third a "3" in the bottom right, etc.

```
text "This is the first page!"

10.times do
  start_new_page
  text "Here comes yet another page."
end

string = "<page>"
odd_options = { at => [bounds.right - 150, 0],
                :width => 150,
                :align => :right,
                :page_filter => :odd,
                :start_count_at => 1 }
even_options = { :at => [0, bounds.left],
                 :width => 150,
                 :align => :left,
                 :page_filter => :even,
                 :start_count_at => 2 }
number_pages string, odd_options
number_pages string, even_options
```

This code snippet was not evaluated inline. You may see its output by running the example file located here:
http://github.com/prawnpdf/prawn/tree/master/manual/repeatable_content/alternate_page_numbering.rb

Security

Security lets you control who can read the document by defining a password.

The examples include:

- How to encrypt the document without the need for a password
- How to configure the regular user permissions
- How to require a password for the regular user
- How to set a owner password that bypass the document permissions

The `encrypt_document` method, as you might have already guessed, is used to encrypt the PDF document.

Once encrypted whoever is using the document will need the user password to read the document. This password can be set with the `:user_password` option. If this is not set the document will be encrypted but a password will not be needed to read the document.

There are some caveats when encrypting your PDFs. Be sure to read the source documentation (you can find it here: https://github.com/prawnpdf/prawn/blob/master/lib/prawn/security.rb) before using this for anything super serious.

Bare encryption. No password needed. Simple password. All permissions granted.

```
# Bare encryption. No password needed.
Prawn::Document.generate("bare_encryption.pdf") do
  text "Bro, no password was asked but the document is still encrypted."
  encrypt_document
end

# Simple password. All permissions granted.
Prawn::Document.generate("simple_password.pdf") do
  text "You was asked for a password."
  encrypt_document(:user_password => "foo", :owner_password => "bar")
end
```

This code snippet was not evaluated inline. You may see its output by running the example file located here:

http://github.com/prawnpdf/prawn/tree/master/manual/security/encryption.rb

security/permissions.rb

Some permissions may be set for the regular user with the following options: `:print_document`, `:modify_contents`, `:copy_contents`, `:modify_annotations`. All this options default to true, so if you'd like to revoke just set them to false.

A user may bypass all permissions if he provides the owner password which may be set with the `:owner_password` option. This option may be set to `:random` so that users will never be able to bypass permissions.

There are some caveats when encrypting your PDFs. Be sure to read the source documentation (you can find it here: https://github.com/prawnpdf/prawn/blob/master/lib/prawn/security.rb) before using this for anything super serious.

User cannot print the document. All permissions revoked and owner password set to random

```ruby
# User cannot print the document.
Prawn::ManualBuilder::Example.generate("cannot_print.pdf") do
  text "If you used the user password you won't be able to print the doc."
  encrypt_document(:user_password => 'foo', :owner_password => 'bar',
                   :permissions => { :print_document => false })
end

# All permissions revoked and owner password set to random
Prawn::ManualBuilder::Example.generate("no_permissions.pdf") do
  text "You may only view this and won't be able to use the owner password."
  encrypt_document(:user_password => 'foo', :owner_password => :random,
                   :permissions => { :print_document     => false,
                                     :modify_contents    => false,
                                     :copy_contents      => false,
                                     :modify_annotations => false })
end
```

This code snippet was not evaluated inline. You may see its output by running the example file located here:

http://github.com/prawnpdf/prawn/tree/master/manual/security/permissions.rb