

Table

Prawn comes with table support out of the box. Tables can be styled in whatever way you see fit. The whole table, rows, columns and cells can be styled independently from each other.

The examples show:

- How to create tables
- What content can be placed on tables
- Subtables (or tables within tables)
- How to style the whole table
- How to use initializer blocks to style only specific portions of the table

Creating tables with Prawn is fairly easy. There are two methods that will create tables for us **table** and **make_table**.

Both are wrappers that create a new **Prawn::Table** object. The difference is that **table** calls the **draw** method after creating the table and **make_table** only returns the created table, so you have to call the **draw** method yourself.

The most simple table can be created by providing only an array of arrays containing your data where each inner array represents one row.

```
t = make_table([ ["this is the first row"],
                ["this is the second row"] ])
t.draw
move_down 20

table([ ["short", "short", "loooooooooooooooooooooong"],
        ["short", "loooooooooooooooooooooong", "short"],
        ["loooooooooooooooooooooong", "short", "short"] ])
```

this is the first row
this is the second row

short	short	loooooooooooooooooooooong
short	loooooooooooooooooooooong	short
loooooooooooooooooooooong	short	short

table/content_and_subtables.rb

There are five kinds of objects which can be put in table cells:

- 1. String: produces a text cell (the most common usage)
- 2. `Prawn::Table::Cell`
- 3. `Prawn::Table`
- 4. Array
- 5. Images

Whenever a table or an array is provided as a cell, a subtable will be created (a table within a cell).

If you'd like to provide a cell or table directly, the best way is to use the `make_cell` and `make_table` methods as they don't call `draw` on the created object.

To insert an image just provide a hash with an with an `:image` key pointing to the image path.


```
cell_1 = make_cell(:content => "this row content comes directly ")
cell_2 = make_cell(:content => "from cell objects")

two_dimensional_array = [ ["..."], ["subtable from an array"], ["..."] ]

my_table = make_table([ ["..."], ["subtable from another table"], ["..."] ])

image_path = "#{Prawn::DATADIR}/images/stef.jpg"

table([ ["just a regular row", "", "", "blah blah blah"],
        [cell_1, cell_2, "", ""],
        ["", "", two_dimensional_array, ""],
        ["just another regular row", "", "", ""],
        [{:image => image_path}, "", my_table, ""]])
```

just a regular row			blah blah blah
this row content comes directly	from cell objects		
		...	
		subtable from an array	
		...	
just another regular row			
		...	
		subtable from another table	
		...	

table/flow_and_header.rb

If the table cannot fit on the current page it will flow to the next page just like free flowing text. If you would like to have the first row treated as a header which will be repeated on subsequent pages set the **:header** option to true.

```
data = [["This row should be repeated on every new page"]]  
data += [["..."]] * 30  
  
table(data, :header => true)
```

[illegible]

This row should be repeated on every new page

...

...

...

...

...

...

...

table/position.rb

The `table()` method accepts a `:position` argument to determine horizontal position of the table within its bounding box. It can be `:left` (the default), `:center`, `:right`, or a number specifying a distance in PDF points from the left side.

```
data = ["The quick brown fox jumped over the lazy dogs. "] * 2

text "Left:"
table data, :position => :left
move_down 10

text "Center:"
table data, :position => :center
move_down 10

text "Right:"
table data, :position => :right
move_down 10

text "100pt:"
table data, :position => 100
```

Left:

The quick brown fox jumped over the lazy dogs.
The quick brown fox jumped over the lazy dogs.

Center:

The quick brown fox jumped over the lazy dogs.
The quick brown fox jumped over the lazy dogs.

Right:

The quick brown fox jumped over the lazy dogs.
The quick brown fox jumped over the lazy dogs.

100pt:

The quick brown fox jumped over the lazy dogs.
The quick brown fox jumped over the lazy dogs.

table/column_widths.rb

Prawn will make its best attempt to identify the best width for the columns. If the end result isn't good, we can override it with some styling.

Individual column widths can be set with the `:column_widths` option. Just provide an array with the sequential width values for the columns or a hash where each key-value pair represents the column 0-based index and its width.

```
data = [ ["this is not quite as long as the others",
         "here we have a line that is long but with smaller words",
         "this is so very loooooooooooooooooooooooooooooooooooooong" ] ]

text "Prawn trying to guess the column widths"
table(data)
move_down 20

text "Manually setting all the column widths"
table(data, :column_widths => [100, 200, 240])
move_down 20

text "Setting only the last column width"
table(data, :column_widths => {2 => 240})
```

Prawn trying to guess the column widths

this is not quite as long as the others	here we have a line that is long but with smaller words	this is so very loooooooooooooooooooooooooooooooooooooong
---	---	---

Manually setting all the column widths

this is not quite as long as the others	here we have a line that is long but with smaller words	this is so very loooooooooooooooooooooooooooooooooooooong
---	---	---

Setting only the last column width

this is not quite as long as the others	here we have a line that is long but with smaller words	this is so very loooooooooooooooooooooooooooooooooooooong
---	---	---

table/width.rb

The default table width depends on the content provided. It will expand up to the current bounding box width to fit the content. If you want the table to have a fixed width no matter the content you may use the **:width** option to manually set the width.

```
text "Normal width:"
table [%w[A B C]]
move_down 20

text "Fixed width:"
table([%w[A B C]], :width => 300)
move_down 20

text "Normal width:"
table([[ "A", "Blah " * 20, "C"]])
move_down 20

text "Fixed width:"
table([[ "A", "Blah " * 20, "C"]], :width => 300)
```

Normal width:

A	B	C
---	---	---

Fixed width:

A	B	C
---	---	---

Normal width:

[illegible]

Fixed width:

[illegible]

table/row_colors.rb

One of the most common table styling techniques is to stripe the rows with alternating colors.

There is one helper just for that. Just provide the `:row_colors` option an array with color values.

```
data = [ ["This row should have one color"],
          ["And this row should have another"] ]

data += [ ["..."] ] * 10

table(data, :row_colors => ["F0F0F0", "FFFFCC"] )
```

[illegible]

table/cell_dimensions.rb

To style all the table cells you can use the `:cell_style` option with the table methods. It accepts a hash with the cell style options.

Some straightforward options are `width`, `height`, and `padding`. All three accept numeric values to set the property.

`padding` also accepts a four number array that defines the padding in a CSS like syntax setting the top, right, bottom, left sequentially. The default is 5pt for all sides.

```
data = [ ["Look at how the cells will look when styled", "", ""],
        ["They probably won't look the same", "", ""]
      ]

{ :width => 160, :height => 50, :padding => 12 }.each do |property, value|
  text "Cell's #{property}: #{value}"
  table(data, :cell_style => {property => value})
  move_down 20
end

text "Padding can also be set with an array: [0, 0, 0, 30]"
table(data, :cell_style => { :padding => [0, 0, 0, 30] })
```

Cell's width: 160

Look at how the cells will look when styled		
They probably won't look the same		

Cell's height: 50

Look at how the cells will look when styled		
They probably won't look the same		

Cell's padding: 12

Look at how the cells will look when styled		
They probably won't look the same		

Padding can also be set with an array: [0, 0, 0, 30]

Look at how the cells will look when styled		
They probably won't look the same		

table/cell_borders_and_bg.rb

The **borders** option accepts an array with the border sides that will be drawn. The default is `[:top, :bottom, :left, :right]`.

border_width may be set with a numeric value.

Both **border_color** and **background_color** accept an HTML like RGB color string ("FF0000")

```
data = [ ["Look at how the cells will look when styled", "", ""],
         ["They probably won't look the same", "", ""]
       ]

{ :borders => [:top, :left],
  :border_width => 3,
  :border_color => "FF0000" }.each do |property, value|

  text "Cell #{property}: #{value.inspect}"
  table(data, :cell_style => {property => value})
  move_down 20
end

text "Cell background_color: FFFFCC"
table(data, :cell_style => {:background_color => "FFFCC"})
```

Cell borders: [:top, :left]

Look at how the cells will look when styled	
They probably won't look the same	

Cell border_width: 3

Look at how the cells will look when styled	
They probably won't look the same	

Cell border_color: "FF0000"

Look at how the cells will look when styled	
They probably won't look the same	

Cell background_color: FFFFCC

Look at how the cells will look when styled	
They probably won't look the same	

table/cell_border_lines.rb

The **border_lines** option accepts an array with the styles of the border sides. The default is `[:solid, :solid, :solid, :solid]`.

border_lines must be set to an array.

```
data = [ ["Look at how the cell border lines can be mixed", "", ""],
         ["dotted top border", "", ""],
         ["solid right border", "", ""],
         ["dotted bottom border", "", ""],
         ["dashed left border", "", ""],
       ]

text "Cell :border_lines => [:dotted, :solid, :dotted, :dashed]"

table(data, :cell_style =>
  { :border_lines => [:dotted, :solid, :dotted, :dashed] })
```

Cell :border_lines => [:dotted, :solid, :dotted, :dashed]

Look at how the cell border lines can be mixed		
dotted top border		
solid right border		
dotted bottom border		
dashed left border		

table/cell_text.rb

Text cells accept the following options: **align**, **font**, **font_style**, **inline_format**, **kerning**, **leading**, **min_font_size**, **overflow**, **rotate**, **rotate_around**, **single_line**, **size**, **text_color**, and **valign**.

Most of these style options are direct translations from the text methods styling options.

```
data = [ [ "Look at how the cells will look when styled", "", "" ],
          [ "They probably won't look the same", "", "" ]
        ]

table data, :cell_style => { :font => "Times-Roman", :font_style => :italic }
move_down 20

table data, :cell_style => { :size => 18, :text_color => "346842" }
move_down 20

table [ [ "Just <font size='18'>some</font> <b><i>inline</i></b>", "", "" ],
        [ "<color rgb='FF00FF'>styles</color> being applied here", "", "" ] ],
      :cell_style => { :inline_format => true }
move_down 20

table [ [ "1", "2", "3", "rotate" ] ], :cell_style => { :rotate => 30 }
move_down 20

table data, :cell_style => { :overflow => :shrink_to_fit, :min_font_size => 8,
                             :width => 60, :height => 30 }
```

Look at how the cells will look when styled		
They probably won't look the same		

Look at how the cells will look when styled		
They probably won't look the same		

Just some <i>inline</i>		
styles being applied here		

1	2	3	rotate
---	---	---	--------

Look at how the cells will		
They probably won't look the		





Prawn can insert images into a table. Just pass a hash into `table()` with an `:image` key pointing to the image.

You can pass the `:scale`, `:fit`, `:position`, and `:vposition` arguments in alongside `:image`; these will function just as in `image()`.

The `:image_width` and `:image_height` arguments set the width/height of the image within the cell, as opposed to the `:width` and `:height` arguments, which set the table cell's dimensions.

```
image = "#{Prawn::DATADIR}/images/prawn.png"

table [
  ["Standard image cell",  { :image => image }],
  [":scale => 0.5",        { :image => image, :scale => 0.5 }],
  [":fit => [100, 200]",   { :image => image, :fit => [100, 200] }],
  [":image_height => 50,
   :image_width => 100",  { :image => image, :image_height => 50,
                           :image_width => 100 }],
  [":position => :center", { :image => image, :position => :center }],
  [":vposition => :center", { :image => image, :vposition => :center,
                              :height => 200 }],
], :width => bounds.width
```

Standard image cell	
:scale => 0.5	
:fit => [100, 200]	
:image_height => 50, :image_width => 100	

:position => :center



:vposition => :center



table/span.rb

Table cells can span multiple columns, rows, or both. When building a cell, use the hash argument constructor with a `:colspan` and/or `:rowspan` argument. Row or column spanning must be specified when building the data array; you can't set the span in the table's initialization block. This is because cells are laid out in the grid before that block is called, so that references to row and column numbers make sense.

Cells are laid out in the order given, skipping any positions spanned by previously instantiated cells. Therefore, a cell with `rowspan: 2` will be missing at least one cell in the row below it. See the code and table below for an example.

It is illegal to overlap cells via spanning. A `Prawn::Errors::InvalidTableSpan` error will be raised if spans would cause cells to overlap.

```
table([
  ["A", { :content => "2x1", :colspan => 2}, "B"],
  [{ :content => "1x2", :rowspan => 2}, "C", "D", "E"],
  [{ :content => "2x2", :colspan => 2, :rowspan => 2}, "F"],
  ["G", "H"]
])
```

A	2x1		B
1x2	C	D	E
	2x2		F
G			H

table/before_rendering_page.rb

Prawn::Table#initialize takes a **:before_rendering_page** argument, to adjust the way an entire page of table cells is styled. This allows you to do things like draw a border around the entire table as displayed on a page.

The callback is passed a Cells object that is numbered based on the order of the cells on the page (e.g., the first row on the page is `cells.row(0)`).

```
table([[ "foo", "bar", "baz" ]] * 40) do |t|
  t.cells.border_width = 1
  t.before_rendering_page do |page|
    page.row(0).border_top_width      = 3
    page.row(-1).border_bottom_width  = 3
    page.column(0).border_left_width   = 3
    page.column(-1).border_right_width = 3
  end
end
```

[illegible]

[illegible]

All of the previous styling options we've seen deal with all the table cells at once.

With initializer blocks we may deal with specific cells. A block passed to one of the table methods (**Prawn::Table.new**, **Prawn::Document#table**, **Prawn::Document#make_table**) will be called after cell setup but before layout. This is a very flexible way to specify styling and layout constraints.

Just like the **Prawn::Document.generate** method, the table initializer blocks may be used with and without a block argument.

The table class has three methods that are handy within an initializer block: **cells**, **rows** and **columns**. All three return an instance of **Prawn::Table::Cells** which represents a selection of cells.

cells return all the table cells, while **rows** and **columns** accept a number or a range as argument which returns a single row/column or a range of rows/columns respectively. (**rows** and **columns** are also aliased as **row** and **column**)

The **Prawn::Table::Cells** class also defines **rows** and **columns** so they may be chained to narrow the selection of cells.

All of the cell styling options we've seen on previous examples may be set as properties of the selection of cells.

```
data = [ [ "Header",          "A " * 5, "B"],
         [ "Data row",       "C",      "D " * 5],
         [ "Another data row", "E",      "F" ] ]

table(data) do
  cells.padding = 12
  cells.borders = []

  row(0).borders      = [ :bottom ]
  row(0).border_width = 2
  row(0).font_style   = :bold

  columns(0..1).borders = [ :right ]

  row(0).columns(0..1).borders = [ :bottom, :right ]
end
```

Header	A A A A A	B
Data row	C	D D D D D
Another data row	E	F

table/filtering.rb

Another way to reduce the number of cells is to **filter** the table.

filter is just like **Enumerable#select**. Pass it a block and it will iterate through the cells returning a new **Prawn::Table::Cells** instance containing only those cells for which the block was not false.

```
data = [ ["Item", "Jan Sales", "Feb Sales"],
         ["Oven", 17, 89],
         ["Fridge", 62, 30],
         ["Microwave", 71, 47]
       ]

table(data) do
  values = cells.columns(1..-1).rows(1..-1)

  bad_sales = values.filter do |cell|
    cell.content.to_i < 40
  end

  bad_sales.background_color = "FFAAAA"

  good_sales = values.filter do |cell|
    cell.content.to_i > 70
  end

  good_sales.background_color = "AAFFAA"
end
```

Item	Jan Sales	Feb Sales
Oven	17	89
Fridge	62	30
Microwave	71	47

We've seen how to apply styles to a selection of cells by setting the individual properties. Another option is to use the **style** method

style lets us define multiple properties at once with a hash. It also accepts a block that will be called for each cell and can be used for some complex styling.

```
table([""] * 8] * 8) do
  cells.style(:width => 24, :height => 24)

  cells.style do |c|
    c.background_color = ((c.row + c.column) % 2).zero? ? '000000' : 'ffffff'
  end
end
```

