Squib Documentation

Release v0.11.0

Andy Meneely

September 15, 2016

Contents

1 Install & Update **3**
- 1.1 Pre-requisites . . . 3
- 1.2 Typical Install . . . 3
- 1.3 Updating Squib . . . 3
- 1.4 OS-Specific Quirks . . . 4

2 Learning Squib **5**
- 2.1 Hello, World! . . . 5
- 2.2 The Squib Way pt 0: Learning Ruby . . . 5
  - 2.2.1 Not a Programmer? . . . 5
  - 2.2.2 What You DON'T Need To Know about Ruby for Squib . . . 6
  - 2.2.3 What You Need to Know about Ruby for Squib . . . 6
  - 2.2.4 Find a good text editor . . . 7
  - 2.2.5 Command line basics . . . 7
  - 2.2.6 Edit-Run-Check. . . . 7
  - 2.2.7 Plan to Fail . . . 8
  - 2.2.8 Ruby Learning Resources . . . 8
- 2.3 The Squib Way pt 1: Zero to Game . . . 9
  - 2.3.1 Prototyping with Squib . . . 9
  - 2.3.2 Get Installed and Set Up . . . 9
  - 2.3.3 Our Idea: Familiar Fights . . . 9
  - 2.3.4 Data or Layout? . . . 9
  - 2.3.5 Initial Card Layout . . . 10
  - 2.3.6 Multiple Cards . . . 11
  - 2.3.7 To the table! . . . 12
  - 2.3.8 Next up... . . . 12
- 2.4 The Squib Way pt 2: Iconography . . . 12
  - 2.4.1 Art: Graphic Design vs. Illustration . . . 13
  - 2.4.2 Iconography in Popular Games . . . 13
  - 2.4.3 How Squib Supports Iconography . . . 14
  - 2.4.4 Back to the Example: Drones vs. Humans . . . 14
  - 2.4.5 Why Ruby+YAML+Spreadsheets Works . . . 15
  - 2.4.6 Icons for Some, But Not All, Cards . . . 16
  - 2.4.7 One Column per Icon . . . 16
- 2.5 The Squib Way pt 3: Workflows . . . 16
- 2.6 Squib + Git . . . 16
- 2.7 Using GameIcons.net . . . 16

3 Parameters are Optional **17**

4 Squib Thinks in Arrays **19**
4.1 Using `range` to specify cards . . . 19
4.2 Behold! The Power of Ruby Arrays . . . 20
4.3 Examples . . . 20
4.4 Contribute Recipes! . . . 20

5 Layouts are Squib's Best Feature **21**
5.1 Order of Precedence for Options . . . 22
5.2 When Layouts Are Similar, Use `extends` . . . 22
5.3 Yes, `extends` is Multi-Generational . . . 24
5.4 Yes, `extends` has Multiple Inheritance . . . 24
5.5 Multiple Layout Files get Merged . . . 24
5.6 Squib Comes with Built-In Layouts . . . 25
5.6.1 fantasy.yml . . . 25
5.6.2 economy.yml . . . 25
5.6.3 tuck_box.yml . . . 25
5.6.4 hand.yml . . . 26
5.6.5 playing_card.yml . . . 26
5.7 See Layouts in Action . . . 26

6 Be Data-Driven with XLSX and CSV **27**
6.1 Hash of Arrays . . . 27
6.2 Quantity Explosion . . . 27

7 Unit Conversion **29**

8 Specifying Colors & Gradients **31**
8.1 Colors . . . 31
8.1.1 by hex-string . . . 31
8.1.2 by name . . . 31
8.1.3 by custom name . . . 31
8.2 Gradients . . . 32
8.3 Examples . . . 32

9 The Mighty text Method **33**
9.1 Fonts . . . 33
9.2 Width and Height . . . 33
9.3 Hints . . . 34
9.4 Extents . . . 34
9.5 Embedding Images . . . 34
9.6 Markup . . . 34
9.7 Examples . . . 34

10 Always Have Bleed **35**

11 Configuration Options **37**
11.1 Options are available as methods . . . 38
11.2 Making Squib Verbose . . . 38

12 Vector vs. Raster Backends **39**

13 Group Your Builds **41**

14 Get Help and Give Help **43**
14.1 Get Help . . . 43
14.2 Help by Troubleshooting . . . 43
14.3 Help by Beta Testing . . . 43
14.3.1 Beta: Using Pre-Builds . . . 44
14.3.2 Beta: About versions . . . 45
14.3.3 Beta: About Bundler+RubyGems . . . 45
14.4 Help by Fixing Bugs . . . 45
14.5 Help by Contributing Code . . . 46

15 DSL Reference **47**
15.1 Squib::Deck.new . . . 47
15.1.1 Options . . . 47
15.1.2 Examples . . . 48
15.2 background . . . 48
15.2.1 Options . . . 48
15.2.2 Examples . . . 48
15.3 build . . . 48
15.3.1 Required Arguments . . . 48
15.3.2 Examples . . . 48
15.4 build_groups . . . 49
15.4.1 Arguments . . . 49
15.4.2 Examples . . . 49
15.5 circle . . . 49
15.5.1 Options . . . 49
15.5.2 Examples . . . 50
15.6 cm . . . 51
15.6.1 Parameters . . . 51
15.6.2 Examples . . . 51
15.7 csv . . . 51
15.7.1 Options . . . 52
15.7.2 Individual Pre-processing . . . 52
15.7.3 Examples . . . 52
15.8 curve . . . 52
15.8.1 Options . . . 52
15.8.2 Examples . . . 54
15.9 disable_build . . . 54
15.9.1 Required Arguments . . . 54
15.9.2 Examples . . . 54
15.10 ellipse . . . 54
15.10.1 Options . . . 54
15.10.2 Examples . . . 55
15.11 enable_build . . . 55
15.11.1 Required Arguments . . . 55
15.11.2 Examples . . . 55
15.12 grid . . . 56
15.12.1 Options . . . 56
15.12.2 Examples . . . 57
15.13 hand . . . 57
15.13.1 Options . . . 57
15.13.2 Examples . . . 57
15.14 hint . . . 57
15.14.1 Options . . . 57
15.14.2 Examples . . . 58

15.15 inches . . . 58
15.15.1 Parameters . . . 58
15.15.2 Examples . . . 58
15.16 line . . . 58
15.16.1 Options . . . 58
15.16.2 Examples . . . 59
15.17 mm . . . 59
15.17.1 Parameters . . . 59
15.17.2 Examples . . . 59
15.18 png . . . 59
15.18.1 Options . . . 59
15.18.2 Examples . . . 61
15.19 polygon . . . 61
15.19.1 Options . . . 61
15.19.2 Examples . . . 62
15.20 rect . . . 62
15.20.1 Options . . . 62
15.20.2 Examples . . . 63
15.21 save . . . 63
15.21.1 Options . . . 63
15.21.2 Examples . . . 63
15.22 save_pdf . . . 64
15.22.1 Options . . . 64
15.22.2 Examples . . . 64
15.23 save_png . . . 64
15.23.1 Options . . . 64
15.23.2 Examples . . . 65
15.24 save_sheet . . . 65
15.24.1 Options . . . 65
15.24.2 Examples . . . 66
15.25 showcase . . . 66
15.25.1 Options . . . 66
15.25.2 Examples . . . 67
15.26 star . . . 67
15.26.1 Options . . . 67
15.26.2 Examples . . . 68
15.27 svg . . . 68
15.27.1 Options . . . 68
15.27.2 Examples . . . 70
15.28 text . . . 70
15.28.1 Options . . . 70
15.28.2 Markup . . . 72
15.28.3 Embedded Icons . . . 73
15.28.4 Examples . . . 75
15.29 triangle . . . 75
15.29.1 Options . . . 75
15.29.2 Examples . . . 76
15.30 use_layout . . . 76
15.30.1 Options . . . 76
15.30.2 Examples . . . 76
15.31 xlsx . . . 76
15.31.1 Options . . . 76
15.31.2 Individual Pre-processing . . . 77
15.31.3 Examples . . . 77

16 Indices and tables 79

Welcome to the official Squib documentation!

Contents:

CHAPTER 1

Install & Update

Squib is a Ruby gem, and installation is handled like most gems.

1.1 Pre-requisites

- Ruby 2.1+

Squib works with both x86 and x86_64 versions of Ruby.

1.2 Typical Install

Regardless of your OS, installation is:

```
$ gem install squib
```

If you're using Bundler, add this line to your application's Gemfile:

```
gem 'squib'
```

And then execute:

```
$ bundle install
```

Squib has some native dependencies, such as Cairo, Pango, and Nokogiri, which will compile upon installation - this is normal.

1.3 Updating Squib

At this time we consider Squib to be still in initial development, so we are not supporting older versions. Please upgrade your Squib as often as possible.

To keep track of when new Squib releases come out, you can watch the BoardGameGeek thread or follow the RSS feed for Squib on its RubyGems page.

In RubyGems, the command looks like this:

```
$ gem up squib
```

As a quirk of Ruby/RubyGems, sometimes older versions of gems get caught in caches. You can see which versions of Squib are installed and clean them up, use `gem list` and `gem cleanup`:

```
$ gem list squib

*** LOCAL GEMS ***

squib (0.9.0, 0.8.0)

$ gem cleanup squib
Cleaning up installed gems...
Attempting to uninstall squib-0.8.0
Successfully uninstalled squib-0.8.0
Clean Up Complete
```

This will remove all prior versions of Squib.

As a sanity check, you can see what version of Squib you're using by referencing the `Squib::VERSION` constant:

```
require 'squib'
puts Squib::VERSION
```

1.4 OS-Specific Quirks

See the wiki for idiosyncracies about specific operating systems, dependency clashes, and other installation issues. If you've run into issues and solved them, please post your solutions for others!

CHAPTER 2

Learning Squib

2.1 Hello, World!

Note: Under construction

2.2 The Squib Way pt 0: Learning Ruby

This guide is for folks who are new to coding and/or Ruby. Feel free to skip it if you already have some coding experience.

2.2.1 Not a Programmer?

> *I'm not a programmer, but I want to use Squib. Can you make it easy for non-programmers?*
>
> —*Frequently Asked Question*

If you want to use Squib, then you want to automate the graphics generation of a tabletop game in a data-driven way. You want to be able to change your mind about icons, illustrations, stats, and graphic design - then rebuild your game with a just a few keystrokes. Essentially, you want to give a list of instructions to the computer, and have it execute your bidding.

If you want those things, then I have news for you. I think you *are* a programmer... who just needs to learn some coding. And maybe Squib will finally be your excuse!

Squib is a Ruby library. To learn Squib, you will need to learn Ruby. There is no getting around that fact. Don't fight it, embrace it.

Fortunately, Squib doesn't really require tons of Ruby-fu to get going. You can really just start from the examples and go from there. And I've done my best to keep to Ruby's own philosophy that programming in it should be a delight, not a chore.

Doubly fortunately,

- Ruby is wonderfully rich in features and expressive in its syntax.
- Ruby has a vibrant, friendly community with people who love to help. I've always thought that Ruby people and board game people would be good friends if they spent more time together.
- Ruby is the language of choice for many new programmers, including many universities.

- Ruby is also "industrial strength", so it really can do just about anything you need it to.

Plus, resources for learning how to code are ubiquitous on the Internet.

In this article, we'll go over some topics that you will undoubtedly want to pick up if you're new to programming or just new to Ruby.

2.2.2 What You DON'T Need To Know about Ruby for Squib

Let's get a few things out of the way. When you are out there searching the interwebs for solutions to your problems, you will *not* need to learn anything about the following things:

- **Rails**. Ruby on Rails is a heavyweight framework for web development. It's awesome in its own way, but it's not relevant to learning Ruby as a language by itself. Squib is about scripting, and will never (NEVER!) be a web app.
- **Object-Oriented Programming**. While OO is very important for developing long-term, scalable applications, some of the philosophy around "Everything in Ruby is an object" can be confusing to newcomers. It's not super-important to grasp this concept for Squib. This means material about classes, modules, mixins, attributes, etc. are not really necessary for Squib scripts. (Contributing to Squib, that's another matter - we use OO a lot internally.)
- **Metaprogramming**. Such a cool thing in Ruby... don't worry about it for Squib. Metaprogramming is for people who literally sleep better at night knowing their designs are extensible for years of software development to come. You're just trying to make a game.

2.2.3 What You Need to Know about Ruby for Squib

I won't give you an introduction to Ruby - other people do that quite nicely (see Resources at the bottom of this article). Instead, as you go through learning Ruby, you should pay special attention to the following:

- Comments
- Variables
- *require*
- What *do* and *end* mean
- Arrays, particularly since most of Squib's arguments are usually Arrays
- Strings and symbols
- String interpolation
- Hashes are important, especially for Excel or CSV importing
- Editing Yaml. Yaml is not Ruby *per se*, but it's a data format common in the Ruby community and Squib uses it in a couple of places (e.g. layouts and the configuration file)
- Methods are useful, but not immediately necessary, for most Squib scripts.

If you are looking for some advanced Ruby-fu, these are useful to brush up on:

- `Enumerable` - everything you can do with iterating over an Array, for example
- `map` - convert one Array to another
- `zip` - combine two arrays in parallel
- `inject` - process one Enumerable and build up something else

2.2.4 Find a good text editor

The text editor is a programmer's most sacred tool. It's where we live, and it's the tool we're most passionate (and dogmatic) about. My personal favorite editors are SublimeText and Atom. There are a bajillion others. The main things you'll need for editing Ruby code are:

- Line numbers. When you get an error, you'll need to know where to go.
- Monospace fonts. Keeping everything lined up is important, especially in keeping indentation.
- Syntax highlighting. You can catch all kinds of basic syntax mistakes with syntax highlighting. My personal favorite syntax highlighting theme is Monokai.
- Manage a directory of files. Not all text editors support this, but Sublime and Atom are particularly good for this (e.g. *Ctrl+P* can open anything!). Squib is more than just the deck.rb - you've got layout files, a config file, your instructions, a build file, and a bunch of other stuff. Your editor should be able to pull those up for you in a few keystrokes so you don't have to go all the way over to your mouse.

There are a ton of other things that these editors will do for you. If you're just starting out, don't worry so much about customizing your editor just yet. Work with it for a while and get used to the defaults. After 30+ hours in the editor, only then should you consider installing plugins and customizing options to fit your preferences.

2.2.5 Command line basics

Executing Ruby is usually done through the command line. Depending on your operating system, you'll have a few options.

- On Macs, you've got the Terminal, which is essentially a Unix shell in Bash (Bourne-Again SHell). This has an amazing amount of customization possible with a long history in the Linux/Unix/BSD world.
- On Windows, there's the Command Prompt (Windows Key, *cmd*). It's a little janky, but it'll do. I've developed Squib primarily in Windows using the Command Prompt.
- If you're on Linux/BSD/etc, you undoubtedly know what the command line is.

For example:

```
$ cd c:\game-prototypes
$ gem install squib
$ squib new tree-gnome-blasters
$ ruby deck.rb
$ rake
$ bundle install
$ gem up squib
```

This might seem arcane at first, but the command line is the single most powerful and expressive tool in computing... if you know how to harness it.

2.2.6 Edit-Run-Check.

To me, the most important word in all of software development is *incremental*. When you're climbing up a mountain by yourself, do you wait to put in anchors until you reach the summit? No!! You anchor yourself along the way frequently so that when you fall, you don't fall very far.

In programming, you need to be running your code often. Very often. In an expressive language like Ruby, you should be running your code every 60-90 seconds (seriously). Why? Because if you make a mistake, then you know that you made it in the last 60-90 seconds, and your problem is that much easier to solve. Solving one bug might take

two minutes, but solving three bugs at once will take ~20 minutes (empirical studies have actually backed this up exponentiation effect).

How much code can you write in 60-90 seconds? Maybe 1-5 lines, for fast typists. Think of it this way: the longer you go without running your code, the more debt you're accruing because it will take longer to fix all the bugs you haven't fixed yet.

That means your code should be stable very often. You'll pick up little tricks here and there. For example, whenever you type a `(`, you should immediately type a `)` afterward and edit in the middle (some text editors even do this for you). Likewise, after every `do` you should type `end` (that's a Ruby thing). There are many, many more. Tricks like that are all about reducing what you have to remember so that you can keep your code stable.

With Squib, you'll be doing one other thing: checking your output. Make sure you have some specific cards to check constantly to make sure the card is coming out the way you want. The Squib method save_png (or ones like it) should be one of the first methods you write when you make a new deck.

As a result of all these, you'll have lots of windows open when working with Squib. You'll have a text editor to edit your source code, your spreadsheet (if you're working with one), a command line prompt, and a preview of your image files. It's a lot of windows, I know. That's why computer geeks usually have multiple monitors!

So, just to recap: your edit-run-check cycle should be *very* short. Trust me on this one.

2.2.7 Plan to Fail

If you get to a point where you can't possibly figure out what's going on that means one thing.

You're human.

Everyone runs into bugs they can't fix. Everyone. Take a break. Put it down. Talk about it out loud. And then, of course, you can always Get Help and Give Help.

2.2.8 Ruby Learning Resources

Here are some of my favorite resources for getting started with Ruby. A lot of them assume you are also new to programming in general. They do cover material that isn't very relevant to Squib, but that's okay - learning is never wasted, only squandered.

CodeSchool's TryRuby This is one of my favorites. It's pretty basic but it walks you through the exercises interactively and makes good use of challenges.

RubyMonk.com An interactive explanation through Ruby. Gets a bit philosophical, but hey, what else would you expect from a monk?

Pragmatic Programmer's Guide to Ruby (The PickAxe Book) One of the best comprehensive resources out there for Ruby - available for free!

Ruby's Own Website: Getting Started This will take you through the basics of programming in Ruby. It works mostly from the Interactive Ruby shell *irb*, which is pretty helpful for seeing how things work and what Ruby syntax looks like.

Why's Poignant Guide to Ruby No list of Ruby resources is complete without a reference to this, well, poignant guide to Ruby. Enjoy.

The Pragmatic Programmer The best software development book ever written (in my opinion). If you are doing programming and you have one book on your shelf, this is it. Much of what inspired Squib came from this thinking.

2.3 The Squib Way pt 1: Zero to Game

Warning: Conversion from markdown not finished yet

2.3.1 Prototyping with Squib

Squib is all about being able to change your mind quickly. Change data, change layout, change artwork, change text. But where do we start? What do we work on first?

The key to prototyping tabletop games is *playtesting*. At the table. With humans. Printed components. That means that we need to get our idea out of our brains and onto pieces of paper as fast as possible.

But! We also want to get the *second* (and third and fourth and fifth...) version of our game back to the playtesting table quickly, too. If we work with Squib from day one, our ability to react to feedback will be much smoother once we've laid the groundwork.

In this series of guides, we'll introduce you to Squib's key features by also walking you through a basic prototype. We'll take a more circuitous route than normal so we can pick apart what Squib is actually doing so that we can leverage its features.

2.3.2 Get Installed and Set Up

The ordinary installation is like most Ruby gems:

```
$ gem install squib
```

See Install & Update for more details.

This guide also assumes you've got some basic Ruby experience, and you've got your tools set up (i.e. text editor, command line, image preview, etc). See The Squib Way pt 0: Learning Ruby to see my recommendations.

2.3.3 Our Idea: Familiar Fights

Let's start with an idea for a game: Familiar Fights. Let's say we want to have players fight each other based on collecting cards that represent their familiars, each with different abilities. We'll have two factions: drones and humans. Each card will have some artwork in it, and some text describing their powers.

First thing: the title. It stinks, I know. It's gonna change. So instead of naming the directory after our game and getting married to our bad idea, let's give our game a code name. I like to use animal names, so let's go with Arctic Lemming:

```
$ squib new arctic-lemming
$ cd arctic-lemming
$ ls
ABOUT.md      Gemfile         PNP NOTES.md    Rakefile        _output         config.yml      deck.rb
```

Go ahead and put "Familiar Fights" as an idea for a title in the `IDEAS.md` file.

If you're using Git or other version control, now's a good time to commit. See Squib + Git.

2.3.4 Data or Layout?

From a prototyping standpoint, we really have two directions we can work from:

- Laying out an example card
- Working on the deck data

There's no wrong direction here - we'll need to do both to get our idea onto the playtesting table. Go where your inspiration guides you. For this example, let's say I've put together ideas for four cards. Here's the data:

name	faction	power
Ninja	human	Use the power of another player
Pirate	human	Steal 1 card from another player
Zombie	drone	Take a card from the discard pile
Robot	drone	Draw two cards

If you're a spreadsheet person, go ahead and put this into Excel (in the above format). Or, if you want to be plaintext-friendly, put it into a comma-separated format (CSV). Like this:

2.3.5 Initial Card Layout

Ok let's get into some code now. Here's an "Hello, World" code snippet

Let's dissect this:

- Line 1: this code will bring in the Squib library for us to use. Keep this at the top.
- Line 2: By convention, we put a blank line between our *require* statements and the rest of our code
- Line 3: Define a new deck of cards. Just 1 card for now
- Line 4: Set the background to pink. Colors can be in various notations - see Specifying Colors & Gradients.
- Line 5: Draw a rectangle around the edge of the deck. Note that this has no arguments, because Parameters are Optional.
- Line 6: Put some text in upper-left the corner of the card.
- Line 7: Save our card out to a png file called `card_00.png`. Ordinarily, this will be saved to `_output/card_00.png`, but in our examples we'll be saving to the current directory (because this documentation has its examples as GitHub gists and gists don't have folders - I do not recommend having `dir: '.'` in your code)

By the way, this is what's created:

Now let's incrementally convert the above snippet into just one of our cards. Let's just focus on one card for now. Later we'll hook it up to our CSV and apply that to all of our cards.

You may have seen in some examples that we can just put in x-y coordinates into our DSL method calls (e.g. `text x: 0, y: 100`). That's great for customizing our work later, but we want to get this to the table quickly. Instead, let's make use of Squib's feature (see Layouts are Squib's Best Feature).

Layouts are a way of specifying some of your arguments in one place - a layout file. The `squib new` command created our own `layout.yml` file, but we can also use one of Squib's built-in layout files. Since we just need a title, artwork, and description, we can just use `economy.yml` (inspired by a popular deck builder that currently has *dominion* over the genre). Here's how that looks:

There are a few key decisions I've made here:

- **Black-and-white**. We're now only using black or white so that we can be printer-friendly.
- **Safe and Cut**. We added two rectangles for guides based on the poker card template from TheGameCrafter.com. This is important to do now and not later. In most print-on-demand templates, we have a 1/8-inch border that is larger than what is to be used, and will be cut down (called a *bleed*). Rather than have to change all our

coordinates later, let's build that right into our prototype. Squib can trim around these bleeds for things like showcase, hand, save_sheet, save_png, and save_pdf. See Always Have Bleed.

- **Title**. We added a title based on our data.
- **layout: 'foo'**. Each command references a "layout" rule. These can be seen in our layout file, which is a built-in layout called `economy.yml` (see ours on GitHub). Later on, we can define our own layout rules in our own file, but for now we just want to get our work done as fast as possible and make use of the stock layout. See Layouts are Squib's Best Feature.

2.3.6 Multiple Cards

Ok now we've got a basic card. But we only have one. The real power of Squib is the ability to customize things *per card*. So if we, say, want to have two different titles on two different cards, our *text* call will look like this:

```
text str: ['Zombie', 'Robot'], layout: 'title'
```

When Squib gets this, it will:

- See that the `str:` option has an array, and put `'Zombie'` on the first card and `'Robot'` on the second.
- See that the `layout:` option is NOT an array - so it will use the same one for every card.

So technically, these two lines are equivalent:

```
text str: ['Zombie', 'Robot'], layout: 'title'
text str: ['Zombie', 'Robot'], layout: ['title','title']
```

Ok back to the game. We COULD just put our data into literal arrays. But that's considered bad programming practice (called *hardcoding*, where you put data directly into your code). Instead, let's make use of our CSV data file.

What the `csv` command does here is read in our file and create a hash of arrays. Each array is a column in the table, and the header to the colum is the key to the hash. To see this in action, check it out on Ruby's interactive shell (`irb`):

```
$ irb
2.1.2 :001 > require 'squib'
 => true
2.1.2 :002 > Squib.csv file: 'data.csv'
 => {"name"=>["Ninja", "Pirate", "Zombie", "Robot"], "class"=>["human", "human", "drone", "drone"], "
```

So, we COULD do this:

```
require 'squib'

Squib::Deck.new cards: 4, layout: 'economy.yml' do
  data = csv file: 'data.csv'
  #rest of our code
end
```

BUT! What if we change the number of total cards in the deck? We won't always have 4 cards (i.e. the number 4 is hardcoded). Instead, let's read in the data outside of our `Squib::Deck.new` and then create the deck size based on that:

```
require 'squib'

data = Squib.csv file: 'data.csv'

Squib::Deck.new cards: data['name'].size, layout: 'economy.yml' do
  #rest of our code
end
```

So now we've got our data, let's replace all of our other hardcoded data from before with their corresponding arrays:

Awesome! Now we've got our all of our cards prototyped out. Let's add two more calls before we bring this to the table:

- `save_pdf` that stitches our images out to pdf
- A version number, based on today's date

The file `_output/output.pdf` gets created now. Note that we *don't* want to print out the bleed area, as that is for the printing process, so we add a 1/8-inch trim (Squib defaults to 300ppi, so 300/8=37.5). The `save_pdf` defaults to 8.5x11 piece of landscape paper, and arranges the cards in rows - ready for you to print out and play!

If you're working with version control, I recommend committing multiple times throughout this process. At this stage, I recommend creating a tag when you are ready to print something out so you know what version precisely you printed out.

2.3.7 To the table!

Squib's job is done, for at least this prototype anyway. Now let's print this sheet out and make some cards!

My recommended approach is to get the following:

- A pack of standard sized sleeves, 2.5"x3.5"
- Some cardstock to give the cards some spring
- A paper trimmer, rotary cutter, knife+steel ruler - some way to cut your cards quickly.

Print your cards out on regular office paper. Cut them along the trim lines. Also, cut your cardstock (maybe a tad smaller than 2.5x3.5) and sleeve them. I will often color-code my cardstock backs in prototypes so I can easily tell them apart. Put the cards into the sleeves. You've got your deck!

Now the most important part: play it. When you think of a rule change or card clarification, just pull the paper out of the sleeve and write on the card. These card print-outs are short-lived anyway.

When you playtest, take copious notes. If you want, you can keep those notes in the PLAYTESTING.md file.

2.3.8 Next up...

We've got a long way to go on our game. We need artwork, iconography, more data, and more cards. We have a lot of directions we could go from here, so in our next guide we'll start looking at a variety of strategies. We'll also look at ways we can keep our code clean and simple so that we're not afraid to change things later on.

2.4 The Squib Way pt 2: Iconography

> **Warning:** This chapter is still being written

In the previous guide, we walked you through the basics of going from ideas in your head to a very simple set of cards ready for playtesting at the table. In this guide we take the next step: creating a visual language.

2.4.1 Art: Graphic Design vs. Illustration

A common piece of advice in the prototyping world is "Don't worry about artwork, just focus on the game and do the artwork later". That's good advice, but a bit over-simplified. What folks usually mean by "artwork" is really "illustration", like the oil painting of a wizard sitting in the middle of the card or the intricate border around the edges.

But games are more than just artwork with text - they're a system of rules that need to be efficiently conveyed to the players. They're a *visual language*. When players are new to your game, the layout of the cards need to facilitate learning. When players are competing in their 30th game, though, they need the cards to maximize usability by reducing their memory load, moving gameplay along efficiently, and provide an overall aesthetic that conveys the game theme. That's what graphic design is all about, and requires a game designer's attention much more than commissioning an illustration.

Developing the visual language on your cards is not a trivial task. It's one that takes a lot of iteration, feedback, testing, improvement, failure, small successes, and reverse-engineering. It's something you should consider in your prototype early on. It's also a series of decisions that don't necessarily require artistic ability - just an intentional effort applied to usability.

Icons and the their placement are, perhaps, the most efficient and powerful tools in your toolbelt for conveying your game's world. In the prototyping process, you don't need to be worried about using icons that are your *final* icons, but you should put some thought into what the visuals will look like because you'll be iterating on that in the design process.

2.4.2 Iconography in Popular Games

When you get a chance, I highly recommend studying the iconography of some popular games. What works for you? What didn't? What kinds of choices did the designers make that works *for their game*? Here are a few that come my mind:

Race for the Galaxy

The majority of the cards in RFTG have no description text on them, and yet the game contains hundreds of unique cards. RFTG has a vast, rich visual iconography that conveys a all of the bonuses and trade-offs of a card efficiently to the user. As a drawback, though, the visual language can be intimidating to new players - every little symbol and icon means a new thing, and sometimes you just need to memorize that "this card does that", until you realize that the icons show that.

But once you know the structure of the game and what various bonuses mean, you can understand new cards very easily. Icons are combined in creative ways to show new bonuses. Text is used only when a bonus is much more complicated than what can be expressed with icons. Icons are primarily arranged along left side of the card so you can hold them in your hand and compare bonuses across cards quickly. All of these design decisions match the gameplay nicely because the game consists of a lot of scrolling through cards in your hand and evaluating which ones you want to play.

Go check out images of Race for the Galaxy on BoardGameGeek.com.

Dominion

Unlike RFTG, Dominion has a much simpler iconography. Most of the bonuses are conveyed in a paragraph of text in the description, with a few classifications conveyed by color or format. The text has icons embedded in it to tie in the concept of Gold, Curses, or Victory Points.

But Dominion's gameplay is much different: instead of going through tons of different cards, you're only playing with 10 piles of cards in front of you. So each game really just requires you to remember what 10 cards mean. Once you purchase a card and it goes into your deck, you don't need to evaluate its worth quickly as in RFTG because you

already bought it. Having most of the game's bonuses in prose means that new bonuses are extremely flexible in their expression. As a result, Dominion's bonuses and iconography is much more about text and identifying known cards than about evaluating new ones.

Go check out images of Dominion on BoardGameGeek.com

2.4.3 How Squib Supports Iconography

Squib is good for supporting any kind of layout you can think of, but it's also good for supporting multiple ways of translating your data into icons on cards. Here are some ways that Squib provides support for your ever-evolving iconography:

- svg method, and all of its features like scaling, ID-specific rendering, direct XML manipulation, and all that the SVG file format has to offer
- png method, and all of its features like blending operators, alpha transparency, and masking
- Layout files allow multiple icons for one data column (see Layouts are Squib's Best Feature)
- Layout files also have the `extends` feature that allows icons to inherit details from each other
- The `range` option on text, svg, and png allows you to specify text and icons for any subset of your files
- The text method allows for embedded icons.
- Ruby provides neat ways of aggregating data with `inject`, `map`, and `zip` that supports iconography

2.4.4 Back to the Example: Drones vs. Humans

Ok, let's go back to our running example, project `arctic-lemming` from Part 1. We created cards for playtesting, but we never put down the faction for each card. That's a good candidate for an icon.

Let's get some stock icons for this exercise. For this example, I went to http://game-icons.net. I set my foreground color to black, and background to white. I then downloaded "auto-repair.svg" and "backup.svg". I'm choosing not to rename the files so that I can find them again on the website if I need to. (If you want to know how to do this process DIRECTLY from Ruby, and not going to the website, check out my *other* Ruby gem called game_icons - it's tailor-made for Squib!)

When we were brainstorming our game, we placed one category of icons in a single column ("faction"). Presumably, one would want the faction icon to be in the same place on every card, but a different icon depending on the card's faction. There are a couple of ways of accomplishing this in Squib. First, here some less-than-clean ways of doing it:

```
svg range: 0, file: 'auto_repair.svg' # hard-coded range number? not flexible
svg range: 1, file: 'auto_repair.svg' # hard-coded range number? not flexible
svg range: 2, file: 'backup.svg'      # hard-coded range number? not flexible
svg range: 3, file: 'backup.svg'      # hard-coded range number? not flexible
# This gets very hard to maintain over time
svg file: ['auto_repair.svg', 'auto_repair.svg', 'backup.svg', 'backup.svg']
# This is slightly easier to maintain, but is more verbose and still hardcoded
svg range: 0..1, file 'auto_repair.svg'
svg range: 2..3, file 'backup.svg'
```

That's too much hardcoding of data into our Ruby code. That's what layouts are for. Now, we've already specified a layout file in our prior example. Fortunately, Squib supports *multiple* layout files, which get combined into a single set of layout styles. So let's do that: we create our own layout file that defines what a `human` is and what a `drone` is. Then just tell `svg` to use the layout data. The data column is simply an array of factions, the icon call is just connecting the factions to their styles with:

```
svg layout: data['faction']
```

So, putting it all together, our code looks like this.

BUT! There's a very important software design principle we're violating here. It's called DRY: Don't Repeat Yourself. In making the above layout file, I hit copy and paste. What happens later when we change our mind and want to move the faction icon!?!? We have to change TWO numbers. Blech.

There's a better way: `extends`

The layout files in Squib also support a special keyword, `extends`, that allows us to "copy" (or "inherit") another style onto our own, and then we can override as we see fit. Thus, the following layout is a bit more DRY:

Much better!

Now if we want to add a new faction, we don't have to copy-pasta any code! We just extend from faction and call in our new SVG file. Suppose we add a new faction that needs a bigger icon - we can define our own `width` and `height` beneath the `extends` that will override the parent values of 75.

Looks great! Now let's get these cards out to the playtesting table!

At this point, we've got a very scalable design for our future iterations. Let's take side-trip and discuss why this design works.

2.4.5 Why Ruby+YAML+Spreadsheets Works

In software design, a "good" design is one where the problem is broken down into a set of easier duties that each make sense on their own, where the interaction between duties is easy, and where to place new responsbilities is obvious.

In Squib, we're using automation to assist the prototyping process. This means that we're going to have a bunch of decisions and responsibilities, such as:

- *Game data decisions*. How many of this card should be in the deck? What should this card be called? What should the cost of this card be?
- *Style Decisions*. Where should this icon be? How big should the font be? What color should we use?
- *Logic Decisions*. Can we build this to a PDF, too? How do we save this in black-and-white? Can we include a time stamp on each card? Can we just save one card this time so we can test quickly?

With the Ruby+YAML+Spreadsheets design, we've separated these three kinds of questions into three areas:

- Game data is in a spreadsheet
- Styles are in YAML layout files
- Code is in Ruby

When you work with this design, you'll probably find yourself spending a lot of time working on one of these files for a long time. That means this design is working.

For example, you might be adjusting the exact location of an image by editing your layout file and re-running your code over and over again to make sure you get the exact x-y coordinates right. That's fine. You're not making game data decisions in that moment, so you shouldn't be presented with any of that stuff. This eases the cognitive complexity of what you're doing.

The best way to preserve this design is to try to keep the Ruby code clean. As wonderful as Ruby is, it's the hardest of the three to edit. It is code, after all. So don't clutter it up with game data or style data - let it be the glue between your styles and your game.

Ok, let's get back to this prototype.

2.4.6 Icons for Some, But Not All, Cards

Note: to be written

2.4.7 One Column per Icon

Note: to be written

2.5 The Squib Way pt 3: Workflows

To be written.

2.6 Squib + Git

Note: To be written

Ideas:

- Workflow
- Tracking binary data (show json method)
- Snippet about "what's changed"
- Releases via tags and versioning

2.7 Using Gamelcons.net

Note: To be written

CHAPTER 3

Parameters are Optional

Squib is all about sane defaults and shorthand specification. Arguments to DSL methods are almost always using hashes, which look a lot like [Ruby 2.0's named parameters](http://www.ruby-doc.org/core-2.0.0/doc/syntax/calling_methods_rdoc.html#label-Keyword+Arguments). This means you can specify your parameters in any order you please. All parameters are optional.

For example *x* and *y* default to 0 (i.e. the upper-left corner of the card). Any parameter that is specified in the command overrides any Squib defaults, *config.yml* settings, or layout rules.

Note: you MUST use named parameters rather than positional parameters. For example: `save :png` will lead to an error like this:

```
C:/Ruby200/lib/ruby/gems/2.0.0/gems/squib-0.0.3/lib/squib/api/save.rb:12:in `save': wrong number of a
    from deck.rb:22:in `block in <main>'
    from C:/Ruby200/lib/ruby/gems/2.0.0/gems/squib-0.0.3/lib/squib/deck.rb:60:in `instance_eval'
    from C:/Ruby200/lib/ruby/gems/2.0.0/gems/squib-0.0.3/lib/squib/deck.rb:60:in `initialize'
    from deck.rb:18:in `new'
    from deck.rb:18:in `<main>'
```

Instead, you must name the parameters: *save format: :png*

Warning: If you provide an option to a DSL method that the DSL method does not recognize, Squib ignores the extraenous option without warning. For example, these two calls have identical behavior:

```
save_png prefix: 'front_'
save_png prefix: 'front_', narf: true
```

CHAPTER 4

Squib Thinks in Arrays

When prototyping card games, you usually want some things (e.g. icons, text) to remain the same across every card, but then other things need to change per card. Maybe you want the same background for every card, but a different title.

The vast majority of Squib's DSL methods can accept two kinds of input: whatever it's expecting, or an array of whatever it's expecting. If it's an array, then Squib expects each element of the array to correspond to a different card.

Think of this like "singleton expansion", where Squib goes "Is this an array? No? Then just repeat it the same across every card". Thus, these two DSL calls are logically equivalent:

```
Squib::Deck.new(cards: 2) do
  text str: 'Hello'
  text str: ['Hello', 'Hello'] # same effect
end
```

But then to use a different string on each card you can do:

```
Squib::Deck.new(cards: 2) do
  text str: ['Hello', 'World']
end
```

Note: Technically, Squib is only looking for something that responds to `each` (i.e. an Enumerable). So whatever you give it should just respond to `each` and it will work as expected.

What if you have an array that doesn't match the size of the deck? No problem - Ruby won't complain about indexing an array out of bounds - it simply returns `nil`. So these are equivalent:

```
Squib::Deck.new(cards: 3) do
  text str: ['Hello', 'Hello']
  text str: ['Hello', 'Hello', nil]  # same effect
end
```

In the case of the text method, giving it an `str: nil` will mean the method won't do anything for that card and move on. Different methods react differently to getting `nil` as an option, however, so watch out for that.

4.1 Using `range` to specify cards

There's another way to limit a DSL method to certain cards: the `range` parameter.

Most of Squib's DSL methods allow a `range` to be specified. This can be an actual Ruby `Range`, or anything that implements `each` (i.e. an Enumerable) that corresponds to the **index** of each card.

Integers are also supported for changing one card only. Negatives work from the back of the deck.

Some quick examples:

```
text range: 0..2  # only cards 0, 1, and 2
text range: [0,2] # only cards 0 and 2
text range: 0     # only card 0
```

4.2 Behold! The Power of Ruby Arrays

One of the more distinctive benefits of Ruby is in its rich set of features for manipulating and accessing arrays. Between `range` and using arrays, you can specify subsets of cards quite succinctly. The following methods in Ruby are particularly helpful:

- Array#each - do something on each element of the array (Ruby folks seldom use for-loops!)
- Array#map - do something on each element of an array and put it into a new array
- Array#select - select a subset of an array
- Enumerable#each_with_index - do something to each element, also being aware of the index
- Enumerable#inject - accumulate elements of an array in a custom way
- Array#zip - combine two arrays, element by element

4.3 Examples

Here are a few recipes for using arrays and ranges in practice:

4.4 Contribute Recipes!

There are a lot more great recipes we could come up with. Feel free to contribute! You can add them here via pull request or via the wiki

CHAPTER 5

Layouts are Squib's Best Feature

Working with tons of options to a method can be tiresome. Ideally everything in a game prototype should be data-driven, easily changed, and your Ruby code should readable without being littered with magic numbers.

For this, most Squib methods have a `layout` option. Layouts are a way of setting default values for any parameter given to the method. They let you group things logically, manipulate options, and use built-in stylings.

Think of layouts and DSL calls like CSS and HTML: you can always specify style in your logic (e.g. directly in an HTML tag), but a cleaner approach is to group your styles together in a separate sheet and work on them separately.

To use a layout, set the `layout:` option on `Deck.new` to point to a YAML file. Any command that allows a `layout` option can be set with a Ruby symbol or string, and the command will then load the specified options. The individual command can also override these options.

For example, instead of this:

```
# deck.rb
Squib::Deck.new do
  rect x: 75, y: 75, width: 675, height: 975
end
```

You can put your logic in the layout file and reference them:

```
# custom-layout.yml
bleed:
  x: 75
  y: 75
  width: 975
  height: 675
```

Then your script looks like this:

```
# deck.rb
Squib::Deck.new(layout: 'custom-layout.yml') do
  rect layout: 'bleed'
end
```

The goal is to make your Ruby code separate the data decisions from logic. For the above example, you are separating the decision to draw rectangle around the "bleed" area, and then your YAML file is defining specifically what "bleed" actually means. (Who is going to remember that `x: 75` means "bleed area"??) This process of separating logic from data makes your code more readable, changeable, and maintainable.

Warning: YAML is very finnicky about not allowing tab characters. Use two spaces for indentation instead. If you get a `Psych` syntax error, this is likely the culprit. Indendation is also strongly enforced in Yaml too. See the Yaml docs for more info.

5.1 Order of Precedence for Options

Layouts will override Squib's system defaults, but are overriden by anything specified in the command itself. Thus, the order of precedence looks like this:

1. Use what the DSL method specified, e.g. `rect x: 25`
2. If anything was not yet specified, use what was given in a layout (if a layout was specified in the command and the file was given to the Deck). e.g. `rect layout: :bleed`
3. If still anything was not yet specified, use what was given in Squib's defaults as defined in the DSL Reference.

For example, back to our example:

```
# custom-layout.yml
bleed:
  x: 0.25in
  y: 0.25in
  width: 2.5in
  height: 3.5in
```

(Note that this example makes use of Unit Conversion)

Combined with this script:

```
# deck.rb
Squib::Deck.new(layout: 'custom-layout.yml') do
  rect layout: 'bleed', x: 50
end
```

The options that go into `rect` will be:

- `x` will be 50 because it's specified in the DSL method and overrides the layout
- `y`, `width`, and `height` were specified in the layout file, so their values are used
- The rect's `stroke_color` (and others options like it) was never specified anywhere, so the default for `rect` is used - as discussed in Parameters are Optional.

Note: Defaults are not global for the name of the option - they are specific to the method itself. For example, the default `fill_color` for `rect` is `'#0000'` but for `showcase` it's `:white`.

Note: Layouts work with *all* options (for DSL methods that support layouts), so you can use options like `file` or `font` or whatever is needed.

Warning: If you provide an option in the Yaml file that is not supported by the DSL method, the DSL method will simply ignore it. Same behavior as described in Parameters are Optional.

5.2 When Layouts Are Similar, Use `extends`

Using layouts are a great way of keeping your Ruby code clean and concise. But those layout Yaml files can get pretty long. If you have a bunch of icons along the top of a card, for example, you're specifying the same `y` option over and over again. This is annoyingly verbose, and what if you want to move all those icons downward at once?

Squib provides a way of reusing layouts with the special *extends'* key. When defining an `'extends` key, we can merge in another key and modify its data coming in if we want to. This allows us to do things like place text next to an icon and be able to move them with each other. Like this:

```
# If we change attack, we move defend too!
attack:
  x: 100
  y: 100
defend:
  extends: attack
  x: 150
  #defend now is {:x => 150, :y => 100}
```

Over time, using `extends` saves you a lot of space in your Yaml files while also giving more structure and readability to the file itself.

You can also **modify** data as they get passed through extends:

```
# If we change attack, we move defend too!
attack:
  x: 100
defend:
  extends: attack
  x: += 50
  #defend now is {:x => 150, :y => 100}
```

The following operators are supported within evaluating `extends`

- += will add the giuven number to the inherited number
- -= will subtract the given number from the inherited number

Both operators also support Unit Conversion

From a design point of view, you can also extract out a base design and have your other layouts extend from them:

```
top_icons:
  y: 100
  font: Arial 36
attack:
  extends: top_icon
  x: 25
defend:
  extends: top_icon
  x: 50
health:
  extends: top_icon
  x: 75
# ...and so on
```

Note: Those fluent in Yaml may notice that `extends` key is similar to Yaml's merge keys. Technically, you can use these together - but I just recommend sticking with `extends` since it does what merge keys do *and more*. If you do choose to use both `extends` and Yaml merge keys, the Yaml merge keys are processed first (upon Yaml parsing), then `extends` (after parsing).

5.3 Yes, `extends` is Multi-Generational

As you might expect, `extends` can be composed multiple times:

```
socrates:
  x: 100
plato:
  extends: socrates
  x: += 10    # evaluates to 150
aristotle:
  extends: plato
  x: += 20    # evaluates to 150
```

5.4 Yes, `extends` has Multiple Inheritance

If you want to extend multiple parents, it looks like this:

```
socrates:
  x: 100
plato:
  y: 200
aristotle:
  extends:
    - socrates
    - plato
  x: += 50    # evaluates to 150
```

If multiple keys override the same keys in a parent, the later ("younger") child in the `extends` list takes precedent. Like this:

```
socrates:
  x: 100
plato:
  x: 200
aristotle:
  extends:
    - plato    # note the order here
    - socrates
  x: += 50     # evaluates to 150 from socrates
```

5.5 Multiple Layout Files get Merged

Squib also supports the combination of multiple layout files. If you provide an `Array` of files then Squib will merge them sequentially. Colliding keys will be completely re-defined by the later file. The `extends` key is processed after *each file*, but can be used across files. Here's an example:

```
# load order: a.yml, b.yml

##############
# file a.yml #
##############
grandparent:
  x: 100
parent_a:
```

```
  extends: grandparent
  x: += 10   # evaluates to 110
parent_b:
  extends: grandparent
  x: += 20   # evaluates to 120

##############
# file b.yml #
##############
child_a:
  extends: parent_a  # i.e. extends a layout in a separate file
  x: += 3    # evaluates to 113 (i.e 110 + 3)
parent_b:    # redefined
  extends: grandparent
  x: += 30   # evaluates to 130 (i.e. 100 + 30)
child_b:
  extends: parent_b
  x: += 3    # evaluates to 133 (i.e. 130 + 3)
```

This can be helpful for:

- Creating a base layout for structure, and one for full color for easier color/black-and-white switching
- Sharing base layouts with other designers

5.6 Squib Comes with Built-In Layouts

Why mess with x-y coordinates when you're first prototyping your game? Just use a built-in layout to get your game to the table as quickly as possible.

If your layout file is not found in the current directory, Squib will search for its own set of layout files. The latest the development version of these can be found on GitHub.

Contributions in this area are particularly welcome!!

The following depictions of the layouts are generated with this script

5.6.1 fantasy.yml

https://github.com/andymeneely/squib/tree/master/lib/squib/layouts/fantasy.yml

5.6.2 economy.yml

https://github.com/andymeneely/squib/tree/master/lib/squib/layouts/economy.yml

5.6.3 tuck_box.yml

Based on TheGameCrafter's template.

https://github.com/andymeneely/squib/tree/master/lib/squib/layouts/tuck_box.yml

5.6.4 hand.yml

https://github.com/andymeneely/squib/tree/master/lib/squib/layouts/hand.yml

5.6.5 playing_card.yml

https://github.com/andymeneely/squib/tree/master/lib/squib/layouts/playing_card.yml

5.7 See Layouts in Action

This sample demonstrates many different ways of using and combining layouts.

This sample demonstrates built-in layouts based on popular games (e.g. `fantasy.yml` and `economy.yml`)

CHAPTER 6

Be Data-Driven with XLSX and CSV

Squib supports importing data from ExcelX (.xlsx) files and Comma-Separated Values (.csv) files. Because Squib Thinks in Arrays, these methods are column-based, which means that they assume you have a header row in your table, and that header row will define the name of the column.

6.1 Hash of Arrays

In both DSL methods, Squib will return a `Hash` of `Arrays` correspoding to each row. Thus, be sure to structure your data like this:

- First row should be a header - preferably with concise naming since you'll reference it in Ruby code
- Rows should represent cards in the deck
- Columns represent data about cards (e.g. "Type", "Cost", or "Name")

Of course, you can always import your game data other ways using just Ruby (e.g. from a REST API, a JSON file, or your own custom format). There's nothing special about Squib's methods in how they relate to `Squib::Deck` other than their convenience.

See xlsx and csv for more details and examples.

6.2 Quantity Explosion

If you want more than one copy of a card, then have a column in your data file called `Qty` and fill it with counts for each card. Squib's xlsx and xlsx methods will automatically expand those rows according to those counts. You can also customize that "Qty" to anything you like by setting the *explode* option (e.g. `explode: 'Quantity'`). Again, see the specific methods for examples.

CHAPTER 7

Unit Conversion

By default, Squib thinks in pixels. This decision was made so that we can have pixel-perfect layouts without automatically scaling everything, even though working in units is sometimes easier. We provide some conversion methods, including looking for strings that end in "in", "cm", or "mm" and computing based on the current DPI. The dpi is set on *Squib::Deck.new* (not *config.yml*).

Example is in *samples/units.rb* found [here](https://github.com/andymeneely/squib/tree/master/samples/units.rb)

CHAPTER 8

Specifying Colors & Gradients

8.1 Colors

8.1.1 by hex-string

You can specify a color via the standard hexadecimal string for RGB (as in HTML and CSS). You also have a few other options as well. You can use:

- 12-bit (3 hex numbers), RGB. e.g. `'#f08'`
- 24-bit (6 hex numbers), RRGGBB. e.g. `'#ff0088'`
- 48-bit (9 hex numbers), RRRGGGBBB. e.g. `'#fff000888'`

Additionally, you can specify the alpha (i.e. transparency) of the color as RGBA. An alpha of `0` is full transparent, and `f` is fully opaque. Thus, you can also use:

- 12-bit (4 hex numbers), RGBA. e.g. `'#f085'`
- 24-bit (8 hex numbers), RRGGBBAA. e.g. `'#ff008855'`
- 48-bit (12 hex numbers), RRRGGGBBBAAA. e.g. `'#fff000888555'`

The `#` at the beginning is optional, but encouraged for readability. In layout files (described in Layouts are Squib's Best Feature), the `#` character will initiate a comment in Yaml. So to specify a color in a layout file, just quote it:

```
# this is a comment in yaml
attack:
  fill_color: '#fff'
```

8.1.2 by name

Under the hood, Squib uses the rcairo color parser to accept around 300 named colors. The full list can be found here.

Names of colors can be either strings or symbols, and case does not matter. Multiple words are separated by underscores. For example, `'white'`, `:burnt_orange`, or `'ALIZARIN_CRIMSON'` are all acceptable names.

8.1.3 by custom name

In your `config.yml`, as described in Configuration Options, you can specify custom names of colors. For example, `'foreground'`.

8.2 Gradients

In most places where colors are allowed, you may also supply a string that defines a gradient. Squib supports two flavors of gradients: linear and radial. Gradients are specified by supplying some xy coordinates, which are relative to the card (not the command). Each stop must be between `0.0` and `1.0`, and you can supply as many as you like. Colors can be specified as above (in any of the hex notations or built-in constant). If you add two or more colors at the same stop, then the gradient keeps the colors in the in order specified and treats it like sharp transition.

The format for linear gradient strings look like this:

```
'(x1,y1)(x2,y2) color1@stop1 color2@stop2'
```

The xy coordinates define the angle of the gradient.

The format for radial gradients look like this:

```
'(x1,y1,radius1)(x2,y2,radius2) color1@stop1 color2@stop2'
```

The coordinates specify an inner circle first, then an outer circle.

In both of these formats, whitespace is ignored between tokens so as to make complex gradients more readable.

If you need something more powerful than these two types of gradients (e.g. mesh gradients), then we suggest encapsulating your logic within an SVG and using the svg method to render it.

8.3 Examples

CHAPTER 9

The Mighty text Method

The text method is a particularly powerful method with a ton of options. Be sure to check the option-by-option details in the DSL reference, but here are the highlights.

9.1 Fonts

To set the font, your `text` method call will look something like this:

```
text str: "Hello", font: 'MyFont Bold 32'
```

The `'MyFont Bold 32'` is specified as a "Pango font string", which can involve a lot of options including backup font families, size, all-caps, stretch, oblique, italic, and degree of boldness. (These options are only available if the underlying font supports them, however.) Here's are some text calls with different Pango font strings:

```
text str: "Hello", font: 'Sans 18'
text str: "Hello", font: 'Arial,Verdana weight=900 style=oblique 36'
text str: "Hello", font: 'Times New Roman,Sans 25'
```

Finally, Squib's `text` method has options such as `font_size` that allow you to override the font string. This means that you can set a blanket font for the whole deck, then adjust sizes from there. This is useful with layouts and `extends` too (see Layouts are Squib's Best Feature).

Note: When the font has a space in the name (e.g. Times New Roman), you'll need to put a backup to get Pango's parsing to work. In some operating systems, you'll want to simply end with a comma:

```
text str: "Hello", font: 'Times New Roman, 25'
```

Note: Most of the font rendering is done by a combination of your installed fonts, your OS, and your graphics card. Thus, different systems will render text slightly differently.

9.2 Width and Height

By default, Pango text boxes will scale the text box to whatever you need, hence the `:native` default. However, for most of the other customizations to work (e.g. center-aligned) you'll need to specify the width. If both the width and the height are specified and the text overflows, then the `ellipsize` option is consulted to figure out what to do with the overflow. Also, the `valign` will only work if `height` is also set to something other than `:native`.

9.3 Hints

Laying out text by typing in numbers can be confusing. What Squib calls "hints" is merely a rectangle around the text box. Hints can be turned on globally in the config file, using the hint method, or in an individual text method. These are there merely for prototyping and are not intended for production. Additionally, these are not to be conflated with "rendering hints" that Pango and Cairo mention in their documentation.

9.4 Extents

Sometimes you want size things based on the size of your rendered text. For example, drawing a rectangle around card's title such that the rectangle perfectly fits. Squib returns the final rendered size of the text so you can work with it afterward. It's an array of hashes that correspond to each card. The output looks like this:

```
Squib::Deck.new(cards: 2) do
  extents = text(str: ['Hello', 'World!'])
  puts extents
end
```

will output:

```
[{:width=>109, :height=>55}, {:width=>142, :height=>55}] # Hello was 109 pixels wide, World 142 pixe
```

9.5 Embedding Images

Squib can embed icons into the flow of text. To do this, you need to define text keys for Squib to look for, and then the corresponding files. The object given to the block is a `TextEmbed`, which supports PNG and SVG. Here's a minimal example:

```
text(str: 'Gain 1 :health:') do |embed|
  embed.svg key: ':health:', file: 'heart.svg'
end
```

9.6 Markup

See *Markup* in text.

9.7 Examples

- Examples of all of the above are crammed into the `text_options.rb` sample found here
- The `embed_text.rb` sample has more examples of embedding text, which can be found here
- The `config_text_markup.rb` sample demonstrates how quoting can be configured, found here

And this one too:

CHAPTER 10

Always Have Bleed

Note: TODO: Under construction

- Always plan to have a printing bleed around the edge of your cards. 1/8 in is standard
- Have a safe zone too
- Layouts make this easy (see the built-in layouts)
- Can use png overlays from templates to make sure it fits
- Trim option is what is used everywhere in Squib
- Trim_radius also lets you show your cards off like how they'll really look

CHAPTER 11

Configuration Options

Squib supports various configuration properties that can be specified in an external file. By default, Squib looks for a file called `config.yml` in the current directory. Or, you can set the `config:` option in `Deck.new` to specify the name of the configuration file.

These properties are intended to be immutable for the life of the Deck, and intended to configure how Squib behaves.

The options include:

progress_bars default: `false`

When set to `true`, long-running operations will show a progress bar in the console

hint default: `:off`

Text hints are used to show the boundaries of text boxes. Can be enabled/disabled for individual commands, or set globally with the *hint* method. This setting is overridden by *hint* (and subsequently individual text).

custom_colors default: `{}`

Defines globally-available named colors available to the deck. Must be specified as a hash in yaml. For example:

```
# config.yml
custom_colors:
  fg: '#abc'
  bg: '#def'
```

antialias default: `'best'`

Set the algorithm that Cairo will use for anti-aliasing throughout its rendering. Available options are `fast`, `good`, `best`, `none`, `gray`, `subpixel`.

Not every option is available on every platform. Using our benchmarks on large decks, *best* is only ~10% slower anyway. For more info see the Cairo docs.

backend default: `'memory'`

Defines how Cairo will store the operations. Can be `svg` or `memory`. See Vector vs. Raster Backends.

prefix default: `'card_'`

When using an SVG backend, cards are auto-saved with this prefix and `'%02d'` numbering format.

img_dir default: `'.'`

For reading image file command (e.g. png and svg), read from this directory instead

warn_ellipsize default: true

Show a warning on the console when text is ellipsized. Warning is issued per card.

warn_png_scale default: true

Show a warning on the console when a PNG file is upscaled. Warning is issued per card.

lsquote default: `"\u2018"`

Smart quoting: change the left single quote when `markup: true`

rsquote default: `"\u2019"`

Smart quoting: change the right single quote when `markup: true`

ldquote default: `"\u201C"`

Smart quoting: change the left double quote when `markup: true`

rdquote default: `"\u201D"`

Smart quoting: change the right double quote when `markup: true`

em_dash default: `"\u2014"`

Convert the `--` to this character when `markup: true`

en_dash default: `"\u2013"`

Convert the `---` to this character when `markup: true`

ellipsis default: `"\u2026"`

Convert `...` to this character when `markup: true`

smart_quotes default: true

When `markup: true`, the `text` method will convert quotes. With `smart_quotes: false`, explicit replacements like em-dashes and en-dashes will be replaced but not smart quotes.

11.1 Options are available as methods

For debugging/sanity purposes, if you want to make sure your configuration options are parsed correctly, the above options are also available as methods within `Squib::Deck`, for example:

```
Squib::Deck.new do
  puts backend # prints 'memory' by default
end
```

11.2 Making Squib Verbose

By default, Squib's logger is set to `WARN`, but more fine-grained logging is embedded in the code. To set the logger, just put this at the top of your script:

```
Squib::logger.level = Logger::INFO
```

If you REALLY want to see tons of output, you can also set DEBUG, but that's not intended for general consumption.

CHAPTER 12

Vector vs. Raster Backends

Squib's graphics rendering engine, Cairo, has the ability to support a variety of surfaces to draw on, including both raster images stored in memory and vectors stored in SVG files. Thus, Squib supports the ability to handle both. They are options in the configuration file `backend: memory` or `backend: svg` described in Configuration Options.

If you use save_pdf then this backend option will determine how your cards are saved too. For *memory*, the PDF will be filled with compressed raster images and be a larger file (yet it will still print at high quality... see discussion below). For SVG backends, PDFs will be smaller. If you have your deck backed by SVG, then the cards are auto-saved, so there is no `save_svg` in Squib. (Technically, the operations are stored and then flushed to the SVG file at the very end.)

There are trade-offs to consider here.

- Print quality is **usually higher** for raster images. This seems counterintuitive at first, but consider where Squib sits in your workflow. It's the final assembly line for your cards before they get printed. Cairo puts *a ton* of work into rendering each pixel perfectly when it works with raster images. Printers, on the other hand, don't think in vectors and will render your paths in their own memory with their own embedded libraries without putting a lot of work into antialiasing and various other graphical esoterica. You may notice that print-on-demand companies such as The Game Crafter only accept raster file types, because they don't want their customers complaining about their printers not rendering vectors with enough care.
- Print quality is **sometimes higher** for vector images, particularly in laser printers. We have noticed this on a few printers, so it's worth testing out.
- PDFs are **smaller** for SVG back ends. If file size is a limitation for you, and it can be for some printers or internet forums, then an SVG back end for vectorized PDFs is the way to go.
- Squib is **greedy** with memory. While I've tested Squib with big decks on older computers, the *memory* backend is quite greedy with RAM. If memory is at a premium for you, switching to SVG might help.
- Squib does **not support every feature** with SVG back ends. There are some nasty corner cases here. If it doesn't, please file an issue so we can look into it. Not every feature in Cairo perfectly translates to SVG.

Note: You can still load PNGs into an SVG-backed deck and load SVGs into a memory-backed deck. To me, the sweet spot is to keep all of my icons, text, and other stuff in vector form for infinite scaling and then render them all to pixels with Squib.

Fortunately, switching backends in Squib is as trivial as changing the setting in the config file (see Configuration Options). So go ahead and experiment with both and see what works for you.

CHAPTER 13

Group Your Builds

Often in the prototyping process you'll find yourself cycling between running your overall build and building a single card. You'll probably be commenting out code in the process.

And even after your code is stable, you'll probably want to build your deck multiple ways: maybe a printer-friendly black-and-white version for print-and-play and then a full color version.

Squib's Build Groups help you with these situations. By grouping your Squib code into different groups, you can run parts of it at a time without having to go back and commenting out code.

Here's a basic example:

Only one group is enabled by default: `:all`. All other groups are disabled by default. To see which groups are enabled currently, the `/dsl/groups` returns the set.

Groups can be enabled and disabled in several ways:

- The `/dsl/enable_group` and `/dsl/disable_group` DSL methods can explicitly enable/disable a group. Again, you're back to commenting out the *enable_group* call, but that's easier than remembering what lines to comment out each time.
- When a `Squib::Deck` is initialized, the environment variable `SQUIB_BUILD` is consulted for a comma-separated string. These are converted to Ruby symbols and the corresponding groups are enabled.

Note that the environment variables are intended to change from run to run, from the command line (see above gist for examples in various OS's).

Note: There should be no need to set the SQUIB_BUILD variable globally on your system.

Don't like how Windows specifies environment variables? One adaptation of this is to do the environment setting in a `Rakefile`. Rake is the build utility that comes with Ruby, and it allows us to set different tasks exactly in this way. This Rakefile works nicely with our above code example:

Thus, you can just run this code with commands like these:

```
$ rake
$ rake pnp
$ rake color
$ rake test
$ rake both
```

CHAPTER 14

Get Help and Give Help

14.1 Get Help

Squib is powerful and customizable, which means it can get complicated pretty quickly. Don't settle for being stuck.

Here's an ordered list of how to find help:

1. Go through this documentation
2. Go through the wiki
3. Go through the samples
4. Google it - people have asked lots of questions about Squib already in many forums.
5. Ask on Stackoverflow using the tags "ruby" and "squib". You will get answers quickly from Ruby programmers it's and a great way for us to archive questions for future Squibbers.
6. Our thread on BoardGameGeek or our guild is quite active and informal (if a bit unstructured).

If you email me directly I'll probably ask you to post your question publicly so we can document answers for future Googling Squibbers.

Please use GitHub issues for bugs and feature requests.

14.2 Help by Troubleshooting

One of the best ways you can help the Squib community is to be active on the above forums. Help people out. Answer questions. Share your code. Most of those forums have a "subscribe" feature.

You can also watch the project on GitHub, which means you get notified when new bugs and features are entered. Try reproducing code on your own machine to confirm a bug. Help write minimal test cases. Suggest workarounds.

14.3 Help by Beta Testing

Squib is a small operation. And programming is hard. So we need testers! In particular, I could use help from people to do the following:

- Test out new features as I write them
- Watch for regression bugs by running their current projects on new Squib code, checking for compatibility issues.

Want to join the mailing list and get notifications? https://groups.google.com/forum/#!forum/squib-testers

There's no time commitment expectation associated with signing up. Any help you can give is appreciated!

14.3.1 Beta: Using Pre-Builds

The preferred way of doing beta testing is by to get Squib directly from my GitHub repository. Bundler makes this easy.

If you are just starting out you'll need to install bundler:

```
$ gem install bundler
```

Then, in the root of your Squib project, create a file called *Gemfile* (capitalization counts). Put this in it:

```
source 'https://rubygems.org'

gem 'squib', git: 'git://github.com/andymeneely/squib', branch: 'master'
```

Then run:

```
$ bundle install
```

Your output will look something like this:

```
Fetching git://github.com/andymeneely/squib
Fetching gem metadata from https://rubygems.org/.........
Fetching version metadata from https://rubygems.org/...
Fetching dependency metadata from https://rubygems.org/..
Resolving dependencies...
Using pkg-config 1.1.6
Using cairo 1.14.3
Using glib2 3.0.7
Using gdk_pixbuf2 3.0.7
Using mercenary 0.3.5
Using mini_portile2 2.0.0
Using nokogiri 1.6.7
Using pango 3.0.7
Using rubyzip 1.1.7
Using roo 2.3.0
Using rsvg2 3.0.7
Using ruby-progressbar 1.7.5
Using squib 0.9.0b from git://github.com/andymeneely/squib (at master)
Using bundler 1.10.6
Bundle complete! 1 Gemfile dependency, 14 gems now installed.
Use `bundle show [gemname]` to see where a bundled gem is installed.
```

To double-check that you're using the test version of Squib, puts this in your code:

```
require 'squib'
puts Squib::VERSION # prints the Squib version to the console when you run this code

# Rest of your Squib code...
```

When you run your code, say `deck.rb`, you'll need to put `bundle exec` in front of it. Otherwise Ruby will just go with full releases (e.g. `0.8` instead of pre-releases, e.g. `0.9a`). That would look like this:

```
$ bundle exec ruby deck.rb
```

If you need to know the exact commit of the build, you can see that commit hash in the generated `Gemfile.lock`. That `revision` field will tell you the *exact* version you're using, which can be helpful for debugging. That will look something like this:

```
remote: git://github.com/andymeneely/squib
  revision: 440a8628ed83b24987b9f6af66ad9a6e6032e781
  branch: master
```

To update to the latest from the repository, run `bundle up`.

To remove Squib versions, run `gem cleanup squib`. This will also remove old Squib releases.

14.3.2 Beta: About versions

- When the version ends in "a" (e.g. `v0.9a`), then the build is "alpha". I could be putting in new code all the time without bumping the version. I try to keep things as stable after every commit, but this is considered the least stable code. (Testing still appreciated here, though.) This is also tracked by my `dev` branch.
- For versions ending in "b" (e.g. `v0.9b`), then the build is in "beta". Features are frozen until release, and we're just looking for bug fixes. This tends to be tracked by the `master` branch in my repository.
- I follow the Semantic Versioning as best I can

14.3.3 Beta: About Bundler+RubyGems

The Gemfile is a configuration file (technically it's a Ruby DSL) for a widely-used library in the Ruby community called Bundler. Bundler is a way of managing multiple RubyGems at once, and specifying exactly what you want.

Bundler is different from RubyGems. Technically, you CAN use RubyGems without Bundler: just `gem install` what you need and your `require` statements will work. BUT Bundler helps you specify versions with the Gemfile, and where to get your gems. If you're switching between different versions of gems (like with being tester!), then Bundler is the way to go. The Bundler website is here: http://bundler.io/.

By convention, your `Gemfile` should be in the root directory of your project. If you did `squib new`, there will be one created by default. Normally, a Squib project Gemfile will look like this. That configuration just pulls the Squib from RubyGems.

But, as a tester, you'll want to have Bundler install Squib from my repository. That would look like this: https://github.com/andymeneely/project-spider-monkey/blob/master/Gemfile. (Just line 4 - ignore the other stuff.) I tend to work with two main branches - dev and master. Master is more stable, dev is more bleeding edge. Problems in the master branch will be a surprise to me, problems in the dev branch probably won't surprise me.

After changing your Gemfile, you'll need to run `bundle install`. That will generate a `Gemfile.lock` file - that's Bundler's way of saying exactly what it's planning on using. You don't modify the Gemfile.lock, but you can look at it to see what version of Squib it's locked onto.

14.4 Help by Fixing Bugs

A great way to make yourself known in the community is to go over our backlog and work on fixing bugs. Even suggestions on troubleshooting what's going on (e.g. trying it out on different OS versions) can be a big help.

14.5 Help by Contributing Code

Our biggest needs are in community support. But, if you happen to have some code to contribute, follow this process:

1. Fork the git repository (https://github.com/[my-github-username]/squib/fork)
2. Create your feature branch (`git checkout -b my-new-feature`)
3. Commit your changes (`git commit -am 'Add some feature'`)
4. Push to the branch (`git push origin my-new-feature`)
5. Create a new Pull Request

Be sure to write tests and samples for new features.

Be sure to run the unit tests and packaging with just `rake`. Also, you can check that the samples render properly with `rake sanity`.

CHAPTER 15

DSL Reference

15.1 Squib::Deck.new

The main interface to Squib. Yields to a block that is used for most of Squib's operations. The majority of the DSL methods are instance methods of `Squib::Deck`.

15.1.1 Options

These options set immutable properties for the life of the deck. They are not intended to be changed in the middle of Squib's operation.

width default: `825`

the width of each card in pixels, including bleed. Supports Unit Conversion (e.g. `'2.5in'`).

height default: `1125`

the height of each card in pixels, including bleed. Supports Unit Conversion (e.g. '3.5in').

cards default: `1`

the number of cards in the deck

dpi default: `300`

the pixels per inch when rendering out to PDF, doing Unit Conversion, or other operations that require measurement.

config default: `'config.yml'`

the file used for global settings of this deck, see Configuration Options. If the file is not found, Squib does not complain.

Note: Since this option has `config.yml` as a default, then Squib automatically looks up a `config.yml` in the current working directory.

---

layout default: `nil`

load a YML file of custom layouts. Multiple files in an array are merged sequentially, redefining collisons in the merge process. If no layouts are found relative to the current working directory, then Squib checks for a built-in layout.

15.1.2 Examples

15.2 background

Fills the background with the given color

15.2.1 Options

All of these options support arrays and singleton expansion (except for **range**). See Squib Thinks in Arrays for deeper explanation.

range default: `:all`

the range of cards over which this will be rendered. See *Using range to specify cards*

color default: `:black`

the color or gradient to fill the background with. See Specifying Colors & Gradients.

15.2.2 Examples

15.3 build

Establish a set of commands that can be enabled/disabled together to allow for customized builds. See Group Your Builds for ways to use this effectively.

15.3.1 Required Arguments

Note: This is an argument, not an option like most DSL methods. See example below.

---

group default: `:all`

The name of the build group. Convention is to use a Ruby symbol.

&block When this group is enabled (and only `:all` is enabled by default), then this block is executed. Otherwise, the block is ignored.

15.3.2 Examples

Use group to organize your Squib code into build groups:

```
Squib::Deck.new do
  build :pnp do
    save_pdf
  end
end
```

15.4 build_groups

Returns the set of group names that have been enabled. See Group Your Builds for ways to use this effectively.

15.4.1 Arguments

(none)

15.4.2 Examples

Use group to organize your Squib code into build groups:

```
Squib::Deck.new do
  enable_build :pnp
  build :pnp do
    save_pdf
  end
  puts build_groups # outputs :all and :pnp
end
```

15.5 circle

Draw a circle centered at the given coordinates

15.5.1 Options

All of these options support arrays and singleton expansion (except for **range**). See Squib Thinks in Arrays for deeper explanation.

x default: `0`

> the x-coordinate to place, relative to the upper-left corner of the card and moving right as it increases. Supports Unit Conversion.

y default: `0`

> the y-coordinate to place, relative to the upper-left corner of the card and moving downward as it increases. Supports Unit Conversion.

radius default: `100`

> radius of the circle. Supports Unit Conversion.

fill_color default: `'#0000'` (fully transparent)

> the color or gradient to fill with. See Specifying Colors & Gradients.

stroke_color default: `:black`

> the color with which to stroke the outside of the shape. See Specifying Colors & Gradients.

stroke_width default: `2`

> the width of the outside stroke. Supports Unit Conversion.

stroke_strategy default: `:fill_first`

Specify whether the stroke is done before (thinner) or after (thicker) filling the shape.

Must be either `:fill_first` or `:stroke_first` (or their string equivalents).

dash default: `''` (no dash pattern set)

Define a dash pattern for the stroke. This is a special string with space-separated numbers that define the pattern of on-and-off alternating strokes, measured in pixels or units. For example, `'0.02in 0.02in'` will be an equal on-and-off dash pattern. Supports Unit Conversion.

cap default: `:butt`

Define how the end of the stroke is drawn. Options are `:square`, `:butt`, and `:round` (or string equivalents of those).

range default: `:all`

the range of cards over which this will be rendered. See *Using range to specify cards*

layout default: `nil`

entry in the layout to use as defaults for this command. See Layouts are Squib's Best Feature.

15.5.2 Examples

Listing 15.1: This snippet and others like it live here

```
require 'squib'

Squib::Deck.new do
  background color: :white

  grid x: 10, y: 10, width: 50,  height: 50,  stroke_color: '#0066FF', stroke_width: 1.5, angle: 0.1
  grid x: 10, y: 10, width: 200, height: 200, stroke_color: '#0066FF', stroke_width: 3, angle: 0.1

  rect x: 305, y: 105, width: 200, height: 50, dash: '4 2'

  rect x: 300, y: 300, width: 400, height: 400,
       fill_color: :blue, stroke_color: :red, stroke_width: 50.0,
       join: 'bevel'

  rect x: 550, y: 105, width: 100, height: 100,
       stroke_width: 5, stroke_color: :orange, angle: -0.2

  ellipse x: 675, y: 105, width: 65, height: 100,
       stroke_width: 5, stroke_color: :orange, angle: -0.2

  circle x: 600, y: 600, radius: 75,
         fill_color: :gray, stroke_color: :green, stroke_width: 8.0

  triangle x1: 50, y1: 50,
           x2: 150, y2: 150,
           x3: 75, y3: 250,
           fill_color: :gray, stroke_color: :green, stroke_width: 3.0

  line x1: 50, y1: 550,
       x2: 150, y2: 650,
       stroke_width: 25.0
```

```
32
33    curve x1: 50,  y1: 850, cx1: 150, cy1: 700,
34          x2: 625, y2: 900, cx2: 150, cy2: 700,
35          stroke_width: 12.0, stroke_color: :cyan,
36          fill_color: :burgundy, cap: 'round'
37
38    ellipse x: 50, y: 925, width: 200, height: 100,
39            stroke_width: 5.0, stroke_color: :cyan,
40            fill_color: :burgundy
41
42    star x: 300, y: 1000, n: 5, inner_radius: 15, outer_radius: 40,
43         fill_color: :cyan, stroke_color: :burgundy, stroke_width: 5
44
45    # default draw is fill-then-stroke. Can be changed to stroke-then-fill
46    star x: 375, y: 1000, n: 5, inner_radius: 15, outer_radius: 40,
47         fill_color: :cyan, stroke_color: :burgundy,
48         stroke_width: 5, stroke_strategy: :stroke_first
49
50    polygon x: 500, y: 1000, n: 5, radius: 25, angle: Math::PI / 2,
51            fill_color: :cyan, stroke_color: :burgundy, stroke_width: 2
52
53    save_png prefix: 'shape_'
54  end
```

15.6 cm

Given centimeters, returns the number of pixels according to the deck's DPI.

15.6.1 Parameters

n the number of centimeters

15.6.2 Examples

```
cm(1)          # 118.11px (for default Deck::dpi of 300)
cm(2) + cm(1)  # 354.33ox (for default Deck::dpi of 300)
```

15.7 csv

Pulls CSV data from .csv files into a hash of arrays keyed by the headers. First row is assumed to be the header row.

Parsing uses Ruby's CSV, with options `{headers: true, converters: :numeric}` http://www.ruby-doc.org/stdlib-2.0/libdoc/csv/rdoc/CSV.html

The `csv` method is a member of `Squib::Deck`, but it is also available outside of the Deck DSL with `Squib.csv()`. This allows a construction like:

```
data = Squib.csv file: 'data.csv'
Squib::Deck.new(cards: data['name'].size) do
end
```

15.7.1 Options

file default: `'deck.csv'`

the CSV-formatted file to open. Opens relative to the current directory. If `data` is set, this option is overridden.

data default: `nil`

when set, CSV will parse this data instead of reading the file.

strip default: `true`

When `true`, strips leading and trailing whitespace on values and headers

explode default: `'qty'`

Quantity explosion will be applied to the column this name. For example, rows in the csv with a `'qty'` of 3 will be duplicated 3 times.

col_sep default: `','`

Column separator. One of the CSV custom options in Ruby. See next option below.

CSV custom options in Ruby standard lib. All of the options in Ruby's std lib version of CSV are supported **except** `headers` is always `true` and `converters` is always set to `:numeric`. See the Ruby Docs for information on the options.

15.7.2 Individual Pre-processing

The `xlsx` method also takes in a block that will be executed for each cell in your data. This is useful for processing individual cells, like putting a dollar sign in front of dollars, or converting from a float to an integer. The value of the block will be what is assigned to that cell. For example:

```
resource_data = Squib.csv(file: 'sample.xlsx') do |header, value|
  case header
  when 'Cost'
    "$#{value}k" # e.g. "3" becomes "$3k"
  else
    value # always return the original value if you didn't do anything to it
  end
end
```

15.7.3 Examples

15.8 curve

Draw a bezier curve using the given coordinates, from x1,y1 to x2,y2. The curvature is set by the control points cx1,cy2 and cx2,cy2.

15.8.1 Options

All of these options support arrays and singleton expansion (except for **range**). See Squib Thinks in Arrays for deeper explanation.

x1 default: `0`

the x-coordinate of the first endpoint. Supports Unit Conversion.

y1 default: `0`

the y-coordinate of the first endpoint. Supports Unit Conversion.

x2 default: `5`

the x-coordinate of the second endpoint. Supports Unit Conversion.

y2 default: `5`

the y-coordinate of the second endpoint. Supports Unit Conversion.

cx1 default: `0`

the x-coordinate of the first control point. Supports Unit Conversion.

cy1 default: `0`

the y-coordinate of the first control point. Supports Unit Conversion.

cx2 default: `5`

the x-coordinate of the second control point. Supports Unit Conversion.

cy2 default: `5`

the y-coordinate of the second control point. Supports Unit Conversion.

fill_color default: `'#0000'` (fully transparent)

the color or gradient to fill with. See Specifying Colors & Gradients.

stroke_color default: `:black`

the color with which to stroke the outside of the shape. See Specifying Colors & Gradients.

stroke_width default: `2`

the width of the outside stroke. Supports Unit Conversion.

stroke_strategy default: `:fill_first`

Specify whether the stroke is done before (thinner) or after (thicker) filling the shape.

Must be either `:fill_first` or `:stroke_first` (or their string equivalents).

dash default: `''` (no dash pattern set)

Define a dash pattern for the stroke. This is a special string with space-separated numbers that define the pattern of on-and-off alternating strokes, measured in pixels or units. For example, `'0.02in 0.02in'` will be an equal on-and-off dash pattern. Supports Unit Conversion.

cap default: `:butt`

Define how the end of the stroke is drawn. Options are `:square`, `:butt`, and `:round` (or string equivalents of those).

range default: `:all`

the range of cards over which this will be rendered. See *Using range to specify cards*

layout default: `nil`

entry in the layout to use as defaults for this command. See Layouts are Squib's Best Feature.

15.8.2 Examples

15.9 disable_build

Disable the given build group for the rest of the build. Thus, code within the corresponding `/dsl/group` block will not be executed. See Group Your Builds for ways to use this effectively.

15.9.1 Required Arguments

build_group_name default: `:all` the name of the group to disable. Convention is to use a Ruby symbol.

15.9.2 Examples

Can be used to disable a group (even if it's enabled via command line):

```
Squib::Deck.new do
  disable_build :pnp
  build :pnp do
    save_pdf
  end
end
```

15.10 ellipse

Draw an ellipse at the given coordinates. An ellipse is an oval that is defined by a bounding rectangle. To draw a circle, see circle.

15.10.1 Options

All of these options support arrays and singleton expansion (except for **range**). See Squib Thinks in Arrays for deeper explanation.

x default: `0`

> the x-coordinate to place, relative to the upper-left corner of the card and moving right as it increases. Supports Unit Conversion.

y default: `0`

> the y-coordinate to place, relative to the upper-left corner of the card and moving downward as it increases. Supports Unit Conversion.

width default: `:deck` (the width of the deck)

> the width of the box. Supports Unit Conversion.

height default: `:deck` (the height of the deck)

> the height of the box. Supports Unit Conversion.

fill_color default: `'#0000'` (fully transparent)

> the color or gradient to fill with. See Specifying Colors & Gradients.

stroke_color default: `:black`

the color with which to stroke the outside of the shape. See Specifying Colors & Gradients.

stroke_width default: `2`

the width of the outside stroke. Supports Unit Conversion.

stroke_strategy default: `:fill_first`

Specify whether the stroke is done before (thinner) or after (thicker) filling the shape.

Must be either `:fill_first` or `:stroke_first` (or their string equivalents).

dash default: `''` (no dash pattern set)

Define a dash pattern for the stroke. This is a special string with space-separated numbers that define the pattern of on-and-off alternating strokes, measured in pixels or units. For example, `'0.02in 0.02in'` will be an equal on-and-off dash pattern. Supports Unit Conversion.

cap default: `:butt`

Define how the end of the stroke is drawn. Options are `:square`, `:butt`, and `:round` (or string equivalents of those).

angle default: 0

the angle at which to rotate the ellipse about it's upper-left corner

range default: `:all`

the range of cards over which this will be rendered. See *Using range to specify cards*

layout default: `nil`

entry in the layout to use as defaults for this command. See Layouts are Squib's Best Feature.

15.10.2 Examples

15.11 enable_build

Enable the given build group for the rest of the build. Thus, code within the corresponding `/dsl/group` block will be executed. See Group Your Builds for ways to use this effectively.

15.11.1 Required Arguments

build_group_name the name of the group to enable. Convention is to use a Ruby symbol.

15.11.2 Examples

Can be used to disable a group (even if it's enabled via command line):

```
Squib::Deck.new do
  disable_build :pnp
  build :pnp do
    save_pdf
  end
end
```

15.12 grid

Draw an unlimited square grid of lines on the deck, starting with x,y and extending off the end of the deck.

15.12.1 Options

All of these options support arrays and singleton expansion (except for **range**). See Squib Thinks in Arrays for deeper explanation.

x default: `0`

the x-coordinate to place, relative to the upper-left corner of the card and moving right as it increases. Supports Unit Conversion.

y default: `0`

the y-coordinate to place, relative to the upper-left corner of the card and moving downward as it increases. Supports Unit Conversion.

width default: `:deck` (the width of the deck)

the spacing between vertical gridlines. Supports Unit Conversion.

height default: `:deck` (the height of the deck)

the spacing between horizontal gridlines. Supports Unit Conversion.

fill_color default: `'#0000'` (fully transparent)

the color or gradient to fill with. See Specifying Colors & Gradients.

stroke_color default: `:black`

the color with which to stroke the outside of the shape. See Specifying Colors & Gradients.

stroke_width default: `2`

the width of the outside stroke. Supports Unit Conversion.

stroke_strategy default: `:fill_first`

Specify whether the stroke is done before (thinner) or after (thicker) filling the shape.

Must be either `:fill_first` or `:stroke_first` (or their string equivalents).

dash default: `''` (no dash pattern set)

Define a dash pattern for the stroke. This is a special string with space-separated numbers that define the pattern of on-and-off alternating strokes, measured in pixels or units. For example, `'0.02in 0.02in'` will be an equal on-and-off dash pattern. Supports Unit Conversion.

cap default: `:butt`

Define how the end of the stroke is drawn. Options are `:square`, `:butt`, and `:round` (or string equivalents of those).

range default: `:all`

the range of cards over which this will be rendered. See *Using range to specify cards*

layout default: `nil`

entry in the layout to use as defaults for this command. See Layouts are Squib's Best Feature.

15.12.2 Examples

15.13 hand

Renders a range of cards fanned out as if in a hand. Saves as PNG regardless of back end.

15.13.1 Options

radius default: `:auto`

The distance from the bottom of each card to the center of the fan. If set to `:auto`, then it is computed as 30% of the card's height. Why 30%? Because it looks good that way. Reasons.

angle_range default: `((Math::PI / -4.0)..(Math::PI / 2))`

The overall width of the fan, in radians. Angle of zero is a vertical card. Further negative angles widen the fan counter-clockwise and positive angles widen the fan clockwise.

margin default: `75`

the margin around the entire image. Supports Unit Conversion.

fill_color default: `:white`

Backdrop color. See Specifying Colors & Gradients.

trim default: `0`

the margin around the card to trim before putting into the image

trim_radius default: `0`

the rounded rectangle radius around the card to trim before putting into the image

file default: `'hand.png'`

The file to save relative to the current directory. Will overwrite without warning.

dir default: `_output`

The directory for the output to be sent to. Will be created if it doesn't exist. Relative to the current directory.

range default: `:all`

the range of cards over which this will be rendered. See *Using range to specify cards*

15.13.2 Examples

15.14 hint

Toggle text hints globally. A text hint is a 1-pixel line drawn around the extents of a text box. They are intended to be temporary guides.

15.14.1 Options

text default: `:off`

The color of the text hint. See Specifying Colors & Gradients To turn off use `:off` or `nil`.

15.14.2 Examples

15.15 inches

Given inches, returns the number of pixels according to the deck's DPI.

15.15.1 Parameters

n the number of inches

15.15.2 Examples

```
inches(2.5)                # 750 (for default Deck::dpi of 300)
inches(2.5) + inches(0.5)  # 900 (for default Deck::dpi of 300)
```

15.16 line

Draw a line from x1,y1 to x2,y2.

15.16.1 Options

All of these options support arrays and singleton expansion (except for **range**). See Squib Thinks in Arrays for deeper explanation.

x1 default: `0`

the x-coordinate to place. Supports Unit Conversion

y1 default: `0`

the y-coordinate to place. Supports Unit Conversion

x2 default: `50`

the x-coordinate to place. Supports Unit Conversion

y2 default: `50`

the y-coordinate to place. Supports Unit Conversion

x3 default: `0`

the x-coordinate to place. Supports Unit Conversion

y3 default: `50`

the y-coordinate to place. Supports Unit Conversion

fill_color default: `'#0000'` (fully transparent)

the color or gradient to fill with. See Specifying Colors & Gradients.

stroke_color default: `:black`

the color with which to stroke the outside of the shape. See Specifying Colors & Gradients.

stroke_width default: `2`

the width of the outside stroke. Supports Unit Conversion.

stroke_strategy default: `:fill_first`

Specify whether the stroke is done before (thinner) or after (thicker) filling the shape.

Must be either `:fill_first` or `:stroke_first` (or their string equivalents).

dash default: `''` (no dash pattern set)

Define a dash pattern for the stroke. This is a special string with space-separated numbers that define the pattern of on-and-off alternating strokes, measured in pixels or units. For example, `'0.02in 0.02in'` will be an equal on-and-off dash pattern. Supports Unit Conversion.

cap default: `:butt`

Define how the end of the stroke is drawn. Options are `:square`, `:butt`, and `:round` (or string equivalents of those).

range default: `:all`

the range of cards over which this will be rendered. See *Using range to specify cards*

layout default: `nil`

entry in the layout to use as defaults for this command. See Layouts are Squib's Best Feature.

15.16.2 Examples

15.17 mm

Given millimeters, returns the number of pixels according to the deck's DPI.

15.17.1 Parameters

n the number of mm

15.17.2 Examples

```
mm(1)          # 11.811px (for default Deck::dpi of 300)
mm(2) + mm(1)  # 35.433ox (for default Deck::dpi of 300)
```

15.18 png

Renders PNG images.

15.18.1 Options

All of these options support arrays and singleton expansion (except for **range**). See Squib Thinks in Arrays for deeper explanation.

file default: `''` (empty string)

file(s) to read in. As in Squib Thinks in Arrays, if this a single file, then it's use for every card in range. If the parameter is an Array of files, then each file is looked up for each card. If any of them are nil or '', nothing is done for that card.

x default: `0`

the x-coordinate to place, relative to the upper-left corner of the card and moving right as it increases. Supports Unit Conversion.

y default: `0`

the y-coordinate to place, relative to the upper-left corner of the card and moving downward as it increases. Supports Unit Conversion.

width default: `:native`

the pixel width that the image should scale to. Supports Unit Conversion. When set to `:native`, uses the DPI and units of the loaded SVG document. Using `:deck` will scale to the deck width. Using `:scale` will use the `height` to scale and keep native the aspect ratio. Scaling PNGs is not recommended for professional-looking cards, and up-scaling a PNG will throw a warning in the console (see Configuration Options).

height default: `:native`

the pixel height that the image should scale to. Supports Unit Conversion. When set to `:native`, uses the DPI and units of the loaded SVG document. Using `:deck` will scale to the deck height. Using `:scale` will use the `width` to scale and keep native the aspect ratio. Scaling PNGs is not recommended for professional-looking cards, and up-scaling a PNG will throw a warning in the console (see Configuration Options).

alpha default: 1.0

the alpha-transparency percentage used to blend this image. Must be between `0.0` and `1.0`

blend default: `:none`

the composite blend operator used when applying this image. See Blend Modes at http://cairographics.org/operators. The possibilties include :none, :multiply, :screen, :overlay, :darken, :lighten, :color_dodge, :color_burn, :hard_light, :soft_light, :difference, :exclusion, :hsl_hue, :hsl_saturation, :hsl_color, :hsl_luminosity. String versions of these options are accepted too.

mask default: `nil`

Accepts a color (see Specifying Colors & Gradients). If specified, the image will be used as a mask for the given color/gradient. Transparent pixels are ignored, opaque pixels are the given color. Note: the origin for gradient coordinates is at the given x,y, not at 0,0 as it is most other places.

angle default: `0`

Rotation of the in radians. Note that this rotates around the upper-left corner, making the placement of x-y coordinates slightly tricky.

crop_x default: `0`

Crop the loaded image at this x coordinate. Supports Unit Conversion.

crop_y default: `0`

Crop the loaded image at this y coordinate. Supports Unit Conversion.

crop_corner_radius default: `0`

Radius for rounded corners, both x and y. When set, overrides crop_corner_x_radius and crop_corner_y_radius. Supports Unit Conversion.

crop_corner_x_radius default: `0`

x radius for rounded corners of cropped image. Supports Unit Conversion.

crop_corner_y_radius default: `0`

y radius for rounded corners of cropped image. Supports Unit Conversion.

crop_width default: `:native`

Width of the cropped image. Supports Unit Conversion.

crop_height default: `:native`

Height of the cropped image. Supports Unit Conversion.

flip_horiztonal default: `false`

Flip this image about its center horizontally (i.e. left becomes right and vice versa).

flip_vertical default: `false`

Flip this image about its center verticall (i.e. top becomes bottom and vice versa).

range default: `:all`

the range of cards over which this will be rendered. See *Using range to specify cards*

layout default: `nil`

entry in the layout to use as defaults for this command. See Layouts are Squib's Best Feature.

15.18.2 Examples

15.19 polygon

Draw an n-sided regular polygon, centered at x,y.

15.19.1 Options

All of these options support arrays and singleton expansion (except for **range**). See Squib Thinks in Arrays for deeper explanation.

x default: `0`

the x-coordinate to place, relative to the upper-left corner of the card and moving right as it increases. Supports Unit Conversion.

y default: `0`

the y-coordinate to place, relative to the upper-left corner of the card and moving downward as it increases. Supports Unit Conversion.

radius default: 0

the distance from the center of the star to the inner circle of its points. Supports Unit Conversion.

angle default: 0

the angle at which to rotate the star

fill_color default: `'#0000'` (fully transparent)

the color or gradient to fill with. See Specifying Colors & Gradients.

stroke_color default: `:black`

the color with which to stroke the outside of the shape. See Specifying Colors & Gradients.

stroke_width default: `2`

the width of the outside stroke. Supports Unit Conversion.

stroke_strategy default: `:fill_first`

Specify whether the stroke is done before (thinner) or after (thicker) filling the shape.

Must be either `:fill_first` or `:stroke_first` (or their string equivalents).

dash default: `''` (no dash pattern set)

Define a dash pattern for the stroke. This is a special string with space-separated numbers that define the pattern of on-and-off alternating strokes, measured in pixels or units. For example, `'0.02in 0.02in'` will be an equal on-and-off dash pattern. Supports Unit Conversion.

cap default: `:butt`

Define how the end of the stroke is drawn. Options are `:square`, `:butt`, and `:round` (or string equivalents of those).

range default: `:all`

the range of cards over which this will be rendered. See *Using range to specify cards*

layout default: `nil`

entry in the layout to use as defaults for this command. See Layouts are Squib's Best Feature.

15.19.2 Examples

15.20 rect

Draw a rounded rectangle

15.20.1 Options

All of these options support arrays and singleton expansion (except for **range**). See Squib Thinks in Arrays for deeper explanation.

x default: `0`

the x-coordinate to place, relative to the upper-left corner of the card and moving right as it increases. Supports Unit Conversion.

y default: `0`

the y-coordinate to place, relative to the upper-left corner of the card and moving downward as it increases. Supports Unit Conversion.

width default: `:deck` (the width of the deck)

the width of the box. Supports Unit Conversion.

height default: `:deck` (the height of the deck)

the height of the box. Supports Unit Conversion.

fill_color default: `'#0000'` (fully transparent)

the color or gradient to fill with. See Specifying Colors & Gradients.

stroke_color default: `:black`

the color with which to stroke the outside of the shape. See Specifying Colors & Gradients.

stroke_width default: `2`

the width of the outside stroke. Supports Unit Conversion.

stroke_strategy default: `:fill_first`

Specify whether the stroke is done before (thinner) or after (thicker) filling the shape.

Must be either `:fill_first` or `:stroke_first` (or their string equivalents).

dash default: `''` (no dash pattern set)

Define a dash pattern for the stroke. This is a special string with space-separated numbers that define the pattern of on-and-off alternating strokes, measured in pixels or units. For example, `'0.02in 0.02in'` will be an equal on-and-off dash pattern. Supports Unit Conversion.

cap default: `:butt`

Define how the end of the stroke is drawn. Options are `:square`, `:butt`, and `:round` (or string equivalents of those).

range default: `:all`

the range of cards over which this will be rendered. See *Using range to specify cards*

layout default: `nil`

entry in the layout to use as defaults for this command. See Layouts are Squib's Best Feature.

angle default: 0

the angle at which to rotate the rectangle about it's upper-left corner

15.20.2 Examples

15.21 save

Saves the given range of cards to either PNG or PDF. Wrapper method for other save methods.

15.21.1 Options

This method delegates everything to save_png or save_pdf using the `format` option. All other options are passed along.

format default: `[]` (do nothing)

Use `:png` to save as a PNG, and `:pdf` to save as PDF. To save to both at once, use `[:png, :pdf]`

15.21.2 Examples

```
save format: :png, prefix: 'front_' # same as: save_png prefix: 'front_'
save format: :pdf, prefix: 'cards_' # same as: save_pdf prefix: 'cards_'
save format: [:png, :pdf]           # same as: save_png; save_pdf
```

15.22 save_pdf

Lays out the cards in range on a sheet and renders a PDF

15.22.1 Options

file default: `'output.pdf'`

the name of the PDF file to save. Will be overwritten without warning.

dir default: `_output`

the directory to save to. Created if it doesn't exist.

width default: `3300`

the height of the page in pixels. Default is 11in * 300dpi. Supports Unit Conversion.

height default: `2550`

the height of the page in pixels. Default is 8.5in * 300dpi. Supports Unit Conversion.

margin default: `75`

the margin around the outside of the page. Supports Unit Conversion.

gap default: `0`

the space in pixels between the cards. Supports Unit Conversion.

trim default: `0`

the space around the edge of each card to trim (e.g. to cut off the bleed margin for print-and-play). Supports Unit Conversion.

15.22.2 Examples

15.23 save_png

Saves the given range of cards to a PNG

15.23.1 Options

range default `:all`

the range of cards over which this will be rendered. See {file:README.md#Specifying_Ranges Specifying Ranges}

dir default: `'_output'`

the directory for the output to be sent to. Will be created if it doesn't exist.

prefix default `'card_'`

the prefix of the file name to be printed.

count_format default: `'%02d'`

the format string used for formatting the card count (e.g. padding zeros). Uses a Ruby format string (see the Ruby doc for `Kernel::sprintf` for specifics)

rotate default: `false`

If `true`, the saved cards will be rotated 90 degrees clockwise. Or, rotate by the number of radians. Intended to rendering landscape instead of portrait. Possible values: `true`, `false`, `:clockwise`, `:counterclockwise`

trim default: `0`

the space around the edge of each card to trim (e.g. to cut off the bleed margin for print-and-play). Supports Unit Conversion.

trim_radius default: `0`

the rounded rectangle radius around the card to trim before saving.

15.23.2 Examples

15.24 save_sheet

Lays out the cards in range and renders a stitched PNG sheet

15.24.1 Options

range default: `:all`

the range of cards over which this will be rendered. See {file:README.md#Specifying_Ranges Specifying Ranges}

columns default: `5`

the number of columns in the grid. Must be an integer

rows default: `:infinite`

the number of rows in the grid. When set to :infinite, the sheet scales to the rows needed. If there are more cards than rows*columns, new sheets are started.

prefix default: `card_`

the prefix of the file name(s)

count_format default: `'%02d'`

the format string used for formatting the card count (e.g. padding zeros). Uses a Ruby format string (see the Ruby doc for `Kernel::sprintf` for specifics)

dir default: `'_output'`

the directory to save to. Created if it doesn't exist.

margin default: `0`

the margin around the outside of the sheet. Supports Unit Conversion.

gap default `0`

the space in pixels between the cards. Supports Unit Conversion.

trim default `0`

the space around the edge of each card to trim (e.g. to cut off the bleed margin for print-and-play). Supports Unit Conversion.

15.24.2 Examples

15.25 showcase

Renders a range of cards in a showcase as if they are sitting in 3D on a reflective surface.

15.25.1 Options

trim default: `0`

the margin around the card to trim before putting into the image

trim_radius default: `0`

the rounded rectangle radius around the card to trim before putting into the image

scale default: `0.8`

Percentage of original width of each (trimmed) card to scale to. Must be between 0.0 and 1.0, but starts looking bad around 0.6.

offset default: `1.1`

Percentage of the scaled width of each card to shift each offset. e.g. 1.1 is a 10% shift, and 0.95 is overlapping by 5%

fill_color default: `:white`

Backdrop color. Usually black or white. See Specifying Colors & Gradients.

reflect_offset default: `15`

The number of pixels between the bottom of the card and the reflection. See Unit Conversion

reflect_strength default: `0.2`

The starting alpha transparency of the reflection (at the top of the card). Percentage between 0 and 1. Looks more realistic at low values since even shiny surfaces lose a lot of light.

reflect_percent default: `0.25`

The length of the reflection in percentage of the card. Larger values tend to make the reflection draw just as much attention as the card, which is not good.

face default: `:left`

which direction the cards face. Anything but `:right` will face left

margin default: `75`

the margin around the entire image. Supports Unit Conversion

fill_color default: `:white`

Backdrop color. Supports Specifying Colors & Gradients.

file default: `'showcase.png'`

The file to save relative to the current directory. Will overwrite without warning.

dir default: `_output`

The directory for the output to be sent to. Will be created if it doesn't exist. Relative to the current directory.

range default: `:all`

the range of cards over which this will be rendered. See *Using range to specify cards*

15.25.2 Examples

15.26 star

Draw an n-pointed star, centered at x,y.

15.26.1 Options

All of these options support arrays and singleton expansion (except for **range**). See Squib Thinks in Arrays for deeper explanation.

x default: `0`

the x-coordinate to place, relative to the upper-left corner of the card and moving right as it increases. Supports Unit Conversion.

y default: `0`

the y-coordinate to place, relative to the upper-left corner of the card and moving downward as it increases. Supports Unit Conversion.

inner_radius default: 0

the distance from the center of the star to the inner circle of its points. Supports Unit Conversion.

outer_radius default: 0

the distance from the center of the star to the outer circle of its points. Supports Unit Conversion.

angle default: 0

the angle at which to rotate the star

fill_color default: `'#0000'` (fully transparent)

the color or gradient to fill with. See Specifying Colors & Gradients.

stroke_color default: `:black`

the color with which to stroke the outside of the shape. See Specifying Colors & Gradients.

stroke_width default: `2`

the width of the outside stroke. Supports Unit Conversion.

stroke_strategy default: `:fill_first`

Specify whether the stroke is done before (thinner) or after (thicker) filling the shape.

Must be either `:fill_first` or `:stroke_first` (or their string equivalents).

dash default: `''` (no dash pattern set)

Define a dash pattern for the stroke. This is a special string with space-separated numbers that define the pattern of on-and-off alternating strokes, measured in pixels or units. For example, `'0.02in 0.02in'` will be an equal on-and-off dash pattern. Supports Unit Conversion.

cap default: `:butt`

Define how the end of the stroke is drawn. Options are `:square`, `:butt`, and `:round` (or string equivalents of those).

range default: `:all`

the range of cards over which this will be rendered. See *Using range to specify cards*

layout default: `nil`

entry in the layout to use as defaults for this command. See Layouts are Squib's Best Feature.

15.26.2 Examples

15.27 svg

Renders an entire svg file at the given location. Uses the SVG-specified units and DPI to determine the pixel width and height. If neither data nor file are specified for a given card, this method does nothing.

Note: Note: if alpha transparency is desired, set that in the SVG.

15.27.1 Options

All of these options support arrays and singleton expansion (except for **range**). See Squib Thinks in Arrays for deeper explanation.

file default: `''` (empty string)

file(s) to read in. As in Squib Thinks in Arrays, if this a single file, then it's use for every card in range. If the parameter is an Array of files, then each file is looked up for each card. If any of them are nil or '', nothing is done for that card.

x default: `0`

the x-coordinate to place, relative to the upper-left corner of the card and moving right as it increases. Supports Unit Conversion.

y default: `0`

the y-coordinate to place, relative to the upper-left corner of the card and moving downward as it increases. Supports Unit Conversion.

range default: `all`

the range of cards over which this will be rendered. See Squib Thinks in Arrays

data default: `nil`

render from an SVG XML string. Overrides `file` if both are specified (a warning is shown).

id default: `nil`

if set, then only render the SVG element with the given id. Prefix '#' is optional. Note: the x-y coordinates are still relative to the SVG document's page.

force_id default: `false`

if set to `true`, then this svg will not be rendered at all if the id is empty or nil. If not set, the entire SVG is rendered. Useful for putting multple icons in a single SVG file.

width default: `native`

the pixel width that the image should scale to. Setting this to `:deck` will scale to the deck height. `:scale` will use the width to scale and keep native the aspect ratio. SVG scaling is done with vectors, so the scaling should be smooth. When set to `:native`, uses the DPI and units of the loaded SVG document.

height default: `:native`

the pixel width that the image should scale to. `:deck` will scale to the deck height. `:scale` will use the width to scale and keep native the aspect ratio. SVG scaling is done with vectors, so the scaling should be smooth. When set to `:native`, uses the DPI and units of the loaded SVG document.

blend default: `:none`

the composite blend operator used when applying this image. See Blend Modes at http://cairographics.org/operators. The possibilties include :none, :multiply, :screen, :overlay, :darken, :lighten, :color_dodge, :color_burn, :hard_light, :soft_light, :difference, :exclusion, :hsl_hue, :hsl_saturation, :hsl_color, :hsl_luminosity. String versions of these options are accepted too.

angle default: `0`

rotation of the image in radians. Note that this rotates around the upper-left corner, making the placement of x-y coordinates slightly tricky.

mask default: `nil`

if specified, the image will be used as a mask for the given color/gradient. Transparent pixels are ignored, opaque pixels are the given color. Note: the origin for gradient coordinates is at the given x,y, not at 0,0 as it is most other places.

crop_x default: `0`

rop the loaded image at this x coordinate. Supports Unit Conversion

crop_y default: `0`

rop the loaded image at this y coordinate. Supports Unit Conversion

crop_corner_radius default: `0`

Radius for rounded corners, both x and y. When set, overrides crop_corner_x_radius and crop_corner_y_radius. Supports Unit Conversion

crop_corner_x_radius default: `0`

x radius for rounded corners of cropped image. Supports Unit Conversion

crop_corner_y_radius default: `0`

y radius for rounded corners of cropped image. Supports Unit Conversion

crop_width default: `0`

width of the cropped image. Supports Unit Conversion

crop_height default: `0`

ive): Height of the cropped image. Supports Unit Conversion

flip_horiztonal default: `false`

Flip this image about its center horizontally (i.e. left becomes right and vice versa).

flip_vertical default: `false`

Flip this image about its center verticall (i.e. top becomes bottom and vice versa).

range default: `:all`

the range of cards over which this will be rendered. See *Using range to specify cards*

layout default: `nil`

entry in the layout to use as defaults for this command. See Layouts are Squib's Best Feature.

15.27.2 Examples

15.28 text

Renders a string at a given location, width, alignment, font, etc.

Unix newlines are interpreted even on Windows (i.e. `"\n"`).

15.28.1 Options

All of these options support arrays and singleton expansion (except for **range**). See Squib Thinks in Arrays for deeper explanation.

str default: `''`

the string to be rendered. Must support `#to_s`.

font default: `'Arial 36'`

the Font description string, including family, styles, and size. (e.g. `'Arial bold italic 12'`). For the official documentation, see the **'Pango docs <http://ruby-gnome2.sourceforge.jp/hiki.cgi?Pango%3A%3AFontDescription#style>)'_**. This description is also quite good.

font_size default: `nil`

an override of font string description (i.e. `font`).

x default: `0`

the x-coordinate to place, relative to the upper-left corner of the card and moving right as it increases. Supports Unit Conversion.

y default: `0`

the y-coordinate to place, relative to the upper-left corner of the card and moving downward as it increases. Supports Unit Conversion.

markup: default: `false`

When set to true, various extra styles are allowed. See *Markup*.

width default: `:auto`

the width of the box the string will be placed in. Stretches to the content by default.. Supports Unit Conversion.

height default: `:auto`

the height of the box the string will be placed in. Stretches to the content by default. Supports Unit Conversion.

wrap default: `:word_char`

when `width` is set, determines the behavior of how the string wraps. The `:word_char` option will break at words, but then fall back to characters when the word cannot fit. Options are `:none`, `:word`, `:char`, `:word_char`. Also: `true` is the same as `:word_char`, `false` is the same as `:none`.

spacing default: `0`

Adjust the spacing when the text is multiple lines. No effect when the text does not wrap.

align default: `:left`

The alignment of the text. [:left, right, :center]

justify default: `false`

toggles whether or not the text is justified or not.

valign default: `:top`

When width and height are set, align text vertically according to the ink extents of the text. [:top, :middle, :bottom]

ellipsize default: `:end`

When width and height are set, determines the behavior of overflowing text. Also: *true* maps to *:end* and *false* maps to *:none*. Default *:end* [:none, :start, :middle, :end, true, false]. Also, as mentioned in Configuration Options, if text is ellipsized a warning is thrown.

angle default: `0`

Rotation of the text in radians. Note that this rotates around the upper-left corner of the text box, making the placement of x-y coordinates slightly tricky.

stroke_width default: `0.0`

the width of the outside stroke. Supports Unit Conversion, see {file:README.md#Units Units}.

stroke_color default: `:black`

the color with which to stroke the outside of the rectangle. {file:README.md#Specifying_Colors___Gradients Specifying Colors & Gradients}

stroke_strategy default: `:fill_first`

specify whether the stroke is done before (thinner) or after (thicker) filling the shape. [:fill_first, :stroke_first]

dash default: `''`

define a dash pattern for the stroke. Provide a string with space-separated numbers that define the pattern of on-and-off alternating strokes, measured in pixels by defautl. Supports Unit Conversion (e.g. `'0.02in 0.02in'`).

hint default: `:nil` (i.e. no hint)

draw a rectangle around the text with the given color. Overrides global hints (see {Deck#hint}).

color default: [String] (:black) the color the font will render to. Gradients supported. See {file:README.md#Specifying_Colors___Gradients Specifying Colors}

fill_color default: `'#0000'` (fully transparent)

the color or gradient to fill with. See Specifying Colors & Gradients.

stroke_color default: `:black`

the color with which to stroke the outside of the shape. See Specifying Colors & Gradients.

stroke_width default: `2`

the width of the outside stroke. Supports Unit Conversion.

stroke_strategy default: `:fill_first`

Specify whether the stroke is done before (thinner) or after (thicker) filling the shape.

Must be either `:fill_first` or `:stroke_first` (or their string equivalents).

dash default: `''` (no dash pattern set)

Define a dash pattern for the stroke. This is a special string with space-separated numbers that define the pattern of on-and-off alternating strokes, measured in pixels or units. For example, `'0.02in 0.02in'` will be an equal on-and-off dash pattern. Supports Unit Conversion.

cap default: `:butt`

Define how the end of the stroke is drawn. Options are `:square`, `:butt`, and `:round` (or string equivalents of those).

range default: `:all`

the range of cards over which this will be rendered. See *Using range to specify cards*

layout default: `nil`

entry in the layout to use as defaults for this command. See Layouts are Squib's Best Feature.

15.28.2 Markup

If you want to do specialized formatting within a given string, Squib has lots of options. By setting `markup: true`, you enable tons of text processing. This includes:

- Pango Markup. This is an HTML-like formatting language that specifies formatting inside your string. Pango Markup essentially supports any formatting option, but on a letter-by-letter basis. Such as: font options, letter spacing, gravity, color, etc. See the Pango docs for details.
- Quotes are converted to their curly counterparts where appropriate.
- Apostraphes are converted to curly as well.
- LaTeX-style quotes are explicitly converted (``` ``like this'' ```)
- Em-dash and en-dash are converted with triple and double-dashes respectively (`--` is an en-dash, and `---` becomes an em-dash.)
- Ellipses can be specified with `...` (three periods). Note that this is entirely different from the `ellipsize` option (which determines what to do with overflowing text).

A few notes:

- Smart quoting assumes the UTF-8 character set by default. If you are in a different character set and want to change how it behaves
- Pango markup uses an XML/HTML-ish processor. Some characters require HTML-entity escaping (e.g. `&` for `&`)

You can also disable the auto-quoting mechanism by setting `smart_quotes: false` in your config. Explicit replacements will still be performed. See Configuration Options

15.28.3 Embedded Icons

The `text` method will also respond to a block. The object that gets passed to this block allows for embedding images into the flow of your text. The following methods are supported:

```
text(str: 'Take 1 :tool: and gain 2 :health:') do |embed|
  embed.svg key: ':tool:', file: 'tool.svg'
  embed.png key: ':health:', file: 'health.png'
end
```

embed.svg

All of these options support arrays and singleton expansion (except for **range**). See Squib Thinks in Arrays for deeper explanation.

key default: `'*'`

the string to replace with the graphic. Can be multiple letters, e.g. `':tool:'`

file default: `''`

file(s) to read in, relative to the root directory or `img_dir` if set in the config.

id default: `nil`

if set, then only render the SVG element with the given id. Prefix '#' is optional. Note: the x-y coordinates are still relative to the SVG document's page.

force_id default: `false`

if set, then this svg will not be rendered at all if the id is empty or nil. If not set, the entire SVG is rendered.

layout default: `nil`

entry in the layout to use as defaults for this command. See Layouts are Squib's Best Feature

width default: `:native`

the width of the image rendered. Does not support `:scale` (yet)

height default: `:native`

the height the height of the image rendered. Does not support `:scale` (yet)

dx default: `0`

"delta x", or adjust the icon horizontally by x pixels

dy default: `0`

"delta y", or adjust the icon vertically by y pixels

flip_horiztonal default: `false`

Flip this image about its center horizontally (i.e. left becomes right and vice versa).

flip_vertical default: `false`

Flip this image about its center verticall (i.e. top becomes bottom and vice versa).

alpha default: `1.0`

the alpha-transparency percentage used to blend this image.

angle default: `0`

rotation of the in radians. Note that this rotates around the upper-left corner, making the placement of x-y coordinates slightly tricky.

embed.png

All of these options support arrays and singleton expansion (except for **range**). See Squib Thinks in Arrays for deeper explanation.

key default: `'*'`

the string to replace with the graphic. Can be multiple letters, e.g. `':tool:'`

file default: `''`

file(s) to read in, relative to the root directory or `img_dir` if set in the config.

layout default: `nil`

entry in the layout to use as defaults for this command. See Layouts are Squib's Best Feature

width default: `:native`

the width of the image rendered.

height default: `:native`

the height the height of the image rendered.

dx default: `0`

"delta x", or adjust the icon horizontally by x pixels

dy default: `0`

"delta y", or adjust the icon vertically by y pixels

flip_horiztonal default: `false`

Flip this image about its center horizontally (i.e. left becomes right and vice versa).

flip_vertical default: `false`

Flip this image about its center verticall (i.e. top becomes bottom and vice versa).

alpha default: `1.0`

the alpha-transparency percentage used to blend this image.

blend default: `:none`

the composite blend operator used when applying this image. See Blend Modes at http://cairographics.org/operators. The possibilties include :none, :multiply, :screen, :overlay, :darken, :lighten, :color_dodge, :color_burn, :hard_light, :soft_light, :difference, :exclusion, :hsl_hue, :hsl_saturation, :hsl_color, :hsl_luminosity. String versions of these options are accepted too.

mask default: `nil`

Accepts a color (see Specifying Colors & Gradients). If specified, the image will be used as a mask for the given color/gradient. Transparent pixels are ignored, opaque pixels are the given color. Note: the origin for gradient coordinates is at the given x,y, not at 0,0 as it is most other places.

angle default: `0`

rotation of the in radians. Note that this rotates around the upper-left corner, making the placement of x-y coordinates slightly tricky.

15.28.4 Examples

15.29 triangle

Draw a triangle at the given coordinates.

15.29.1 Options

All of these options support arrays and singleton expansion (except for **range**). See Squib Thinks in Arrays for deeper explanation.

x1 default: `100`

the first x-coordinate to place. Supports Unit Conversion

y1 default: `100`

the first y-coordinate to place. Supports Unit Conversion

x2 default: `150`

the second x-coordinate to place. Supports Unit Conversion

y2 default: `150`

the second y-coordinate to place. Supports Unit Conversion

x3 default: `100`

the third x-coordinate to place. Supports Unit Conversion

y3 default: `150`

the third y-coordinate to place. Supports Unit Conversion

fill_color default: `'#0000'` (fully transparent)

the color or gradient to fill with. See Specifying Colors & Gradients.

stroke_color default: `:black`

the color with which to stroke the outside of the shape. See Specifying Colors & Gradients.

stroke_width default: `2`

the width of the outside stroke. Supports Unit Conversion.

stroke_strategy default: `:fill_first`

Specify whether the stroke is done before (thinner) or after (thicker) filling the shape.

Must be either `:fill_first` or `:stroke_first` (or their string equivalents).

dash default: `''` (no dash pattern set)

Define a dash pattern for the stroke. This is a special string with space-separated numbers that define the pattern of on-and-off alternating strokes, measured in pixels or units. For example, `'0.02in 0.02in'` will be an equal on-and-off dash pattern. Supports Unit Conversion.

cap default: `:butt`

Define how the end of the stroke is drawn. Options are `:square`, `:butt`, and `:round` (or string equivalents of those).

range default: `:all`

the range of cards over which this will be rendered. See *Using range to specify cards*

layout default: `nil`

entry in the layout to use as defaults for this command. See Layouts are Squib's Best Feature.

15.29.2 Examples

15.30 use_layout

Load a layout file and merge into the current set of layouts.

15.30.1 Options

file default: `'layout.yml'`

The file or array of files to load. Treated exactly how Squib::Deck.new parses it.

15.30.2 Examples

15.31 xlsx

Pulls ExcelX data from .xlsx files into a hash of arrays keyed by the headers. First row is assumed to be the header row.

The `xlsx` method is a member of `Squib::Deck`, but it is also available outside of the Deck DSL with `Squib.xlsx()`. This allows a construction like:

```
data = Squib.xlsx file: 'data.xlsx'
Squib::Deck.new(cards: data['name'].size) do
end
```

15.31.1 Options

file default: `'deck.xlsx'`

the xlsx-formatted file to open. Opens relative to the current directory.

sheet default: `0`

The zero-based index of the sheet from which to read.

strip default: `true`

When `true`, strips leading and trailing whitespace on values and headers

explode default: `'qty'`

Quantity explosion will be applied to the column this name. For example, rows in the csv with a `'qty'` of 3 will be duplicated 3 times.

15.31.2 Individual Pre-processing

The `xlsx` method also takes in a block that will be executed for each cell in your data. This is useful for processing individual cells, like putting a dollar sign in front of dollars, or converting from a float to an integer. The value of the block will be what is assigned to that cell. For example:

```
resource_data = Squib.xlsx(file: 'sample.xlsx') do |header, value|
  case header
  when 'Cost'
    "$#{value}k" # e.g. "3" becomes "$3k"
  else
    value # always return the original value if you didn't do anything to it
  end
end
```

15.31.3 Examples

CHAPTER 16

Indices and tables

- genindex
- search