Squib Documentation

Release v0.13.1

Andy Meneely

Apr 16, 2017

Contents

Welcome to the official Squib documentation!

Contents:

CHAPTER 1

Install & Update

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

Pre-requisites

- Ruby 2.1+

Squib works with both x86 and x86_64 versions of Ruby.

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.

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
```

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

Hello, World! Dissected

After seeing Squib's landing page, your might find it helpful to dissect what's really going on in each line of code of a basic Squib snippet.

```
1 require 'squib'
2
3 Squib::Deck.new width: 825, height: 1125, cards: 2 do
4   background color: 'pink'
5   rect
6   text str: ['Hello', 'World!']
7   save_png prefix: 'hello_'
8 end
```

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 2 cards. 825 pixels wide etc. Squib also supports *Unit Conversion* if you prefer to specify something like `'2.75in'`.
- Line 4: Set the background to pink. Colors can be in various notations, and supports linear and radial graidents - see *Specifying Colors & Gradients*.
- Line 5: Draw a rectangle around the edge of each card. Note that this has no arguments, because *Parameters are Optional*. The defaults can be found in the DSL reference for the *rect* method.
- Line 6: Put some text in upper-left the corner of the card, using the default font, etc. See the *text* DSL method for more options. The first card will have `'Hello'` and the second card will have `'World'` because *Squib Thinks in Arrays*.
- Line 7: Save our card out to a png files called `hello_00.png` and `hello_01.png` saved in the `_output` folder.

Dissection of "Even Bigger..."

On Squib's landing page we end with a pretty complex example. It's compact and designed to show how much you can get done with a little bit of code. Here's a dissection of that.

```
1 require 'squib'
2 
3 Squib::Deck.new(cards: 4, layout: %w(hand.yml even-bigger.yml)) do
4   background color: '#230602'
5   deck = xlsx file: 'even-bigger.xlsx'
6   svg file: deck['Art'], layout: 'Art'
7 
8   %w(Title Description Snark).each do |key|
9     text str: deck[key], layout: key
10   end
11 
12   %w(Attack Defend Health).each do |key|
13     svg file: "#{key.downcase}.svg", layout: "#{key}Icon"
14     text str: deck[key], layout: key
15   end
16 
17   save_png prefix: 'even_bigger_'
18   showcase file: 'showcase.png', fill_color: '#0000'
19   hand file: 'hand.png', trim: 37.5, trim_radius: 25, fill_color: '#0000'
20 end
```

- Line 3: Make 4 cards. Use two layouts: the built-in hand.yml (see *Layouts are Squib's Best Feature*) and then our own layout. The layouts get merged, with our own *even-bigger.yml* overriding `hand.yml` whenever they collide.
- Line 5: Read some data from an Excel file, which amounts to a column-based hash of arrays, so that each element of an array corresponds to a specific data point to a given card. For example, `3` in the `'Attack'` column will be put on the second card.
- Line 6: Using the Excel data cell for the filename, we can customize a different icon for every card. But, every SVG in this command will be styled according to the `Art` entry in our layout (i.e. in `even-bigger.yml`)
- Line 8: Iterate over an array of strings, namely, `'Title'`, `'Description'`, and `'Snark'`.
- Line 9: Draw text for the (Title, Description, or Snark), using *their* styling rules in our layout.
- Line 13: Using Ruby String interpolation, lookup the appropriate icon (e.g. `'attack.svg'`), converted to lowercase letters, and then using the Icon layout of that for styling (e.g. `'AttackIcon'` or `'DefendIcon'`)
- Line 17: Render every card to individual PNG files
- Line 18: Render a "showcase" of cards, using a perspective-reflect effect. See *showcase* method.
- Line 19: Render a "hand" of cards (spread over a circle). See *hand* method.

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.

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.

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.

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

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.

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.

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.

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*.

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.

The Squib Way pt 1: Zero to Game

I've always felt that the Ruby community and the tabletop game design community had a lot in common, and a lot to learn from each other. Both are all about testing. All about iterative development. Both communities are collegial, creative, and fun.

But the Ruby community, and the software development community generally, has a lot to teach us game designers about how to develop something. Ruby has a "way" of doing things that is unique and helpful to game designers.

In this series of guides, I'll introduce you to Squib's key features and I'll walk you through a basic prototype. We'll also take a more circuitous route than normal so that I can touch upon some key design principles and good software development habits so that you can make your Squib scripts maintainable, understandable, flexible, and changeable.

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.

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.

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          layout.yml
```

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*.

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:

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*.

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"], "power"=>["Use the power of another player", "Steal 1 card from␣
↪another player", "Take a card from the discard pile", "Draw two cards"]}
```

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.

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.

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.

The Squib Way pt 2: Iconography

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.

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.

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

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 cards
- The *text* method allows for embedded icons.
- The *text* method allows for Unicode characters (if the font supports it).
- Ruby provides neat ways of aggregating data with `inject`, `map`, and `zip` that gives you ultimate flexibility for specifying different icons for different cards.

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! We've got some documentation in *Squib + Game-Icons.net*

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.

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 responsibilities 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.

Illustration: One per Card

The cards are starting to come together, but we have another thing to do now. When playtesting, you need a way of visually identifying the card immediately. Reading text takes an extra moment to identify the card - wouldn't it be nice if we had some sort of artwork, individualized to the card?

Of course, we're not going to commission an artist or do our own oil paintings just yet. Let's get some placeholder art in there. Back to GameIcons, we're going to use "ninja-mask.svg", "pirate-skull.svg", "shambling-zombie.svg", and "robot-golem.svg".

Method 1: Put the file name in data

The difference between our Faction icon and our Illustration icon is that the Illustration needs to be different for every card. We already have a convenient way to do something different on every card - our CSV file!

Here's how the CSV would look:

In our layout file we can center it in the middle of the card, nice and big. And then the Ruby & YAML would look like this:

And our output will look like this:

Method 2: Map title to file name

There are some drawbacks to Method 1. First, you're putting artwork graphics info inside your game data. This can be weird and unexpected for someone new to your code (i.e. that person being you when you put your project down for a few months). Second, when you're working on artwork you'll have to look up what the name of every file is in your CSV. (Even writing this tutorial, I forgot that "zombie" is called "shambling-zombie.svg" and had to look it up, distracting me from focusing on writing.)

There's another way of doing this, and it's more Ruby-like because it follows the Convention over Configuration philosophy. The idea is to be super consistent with your naming so that you don't have to *configure* that, say, "pirate" has an illustration "pirate-skull". The illustration should be literally the title of the card - converted to lowercase because that's the convention for files. That means it should just be called "pirate.svg", and Squib should know to "just put an SVG that is named after the title". Months later, when you want to edit the illustration for pirate, you will probably just open "pirate.svg".

To do this, you'll need to convert an array of Title strings from your CSV (`data['title']` to an array of file names. Ruby's `map` was born for this.

Note: If you're new to Ruby, here's a quick primer. The `map` method gets run on every element of an array, and it lets you specify a *block* (either between curly braces for one line or between `do` and `end` for multiple lines). It then returns another Array of the same size, but with every value mapped using your block. So:

```
[1, 2, 3].map { |x| 2 * x }             # returns [2, 4, 6]
[1, 2, 3].map { |x| "$#{x}" }           # returns ["$1", "$2", "$3"]
['NARF', 'ZORT'].map { |x| x.downcase } # returns ['narf', 'zort']
```

Thus, if we rename our illustration files from "pirate-skull.svg" to "pirate.svg", we can have CSV data that's JUST game data:

And our Ruby code will figure out the file name:

And our output images look identical to Method 1.

Learn by Example

In my game, Your Last Heist, I use some similar methods as above:

- Use a different background depending on if the character is level 1 or 2. Makes use of Ruby's ternary operator.
- Only put an image if the data is non-nil. Some characters have a third skill, others do not. Only load a third skill image if they have a third skill. This line leverages the fact that when you do `svg file: nil`, the `svg` simply does nothing.
- Method 2 from above, but into its own directory.
- Use different-sized backdrops depending on the number of letters in the text. This one is cool because I can rewrite the description of a card, and it will automatically figure out which backdrop to use based on how many letters the text has. This makes use of Ruby's case-when expression.
- After saving the regular cards, we end our script by creating some annotated figures for the rulebook by drawing some text on top of it and saving it using `showcase`.

Are We Done?

At this stage, you've got most of what you need to build a game from prototype through completion. Between images and text, you can do pretty much anything. Squib does much more, of course, but these are the basic building blocks.

But, prototyping is all about speed and agility. The faster you can get what you need, the sooner you can playtest, and the sooner you can make it better. Up next, we'll be looking at workflow: ways to help you get what you need quickly and reliably.

The Squib Way pt 3: Workflows

Warning: Under construction

As we mentioned at the end *The Squib Way pt 2: Iconography*, we've pretty much got most of what we need to prototype a game through completion. From here on out, the *DSL Reference* will be your best resource for getting things done. Everything from here one out is optional, but useful.

But, as you explore Squib's features and work away at your games, you'll pick up a few tricks and conventions to follow that makes your time easier. After years of developing games with Squib, here are some helpful ways of improving your workflow.

Improving your workflow comes down to a few principles:

- **Automate what will be tedious**. There's a balance here. What do you anticipate will change about your game as you develop it? What do you anticipate will *not* change? If you automate *everything*, you will probably spend more time on automating than game development. If you don't automate anything, you'll be re-making every component whenever you make a game design change.
- **Focus on one thing only: visual, game, or build**. Cognitively, you'll have an easier time when you focus on one thing and one thing only. The more loose ends you need to keep in your head, the more mistakes you'll make.

Additionally, improving your workflow can help you pivot to other tasks you might need for polishing your game later on, such as:

- Quickly building one card at a time to reduce the time between builds
- Maintaining a printer-friendly, black-and-white version of your game in tandem with a color version
- Building annotated figures for your rulebook
- Rolling back to older versions of your game

Organizing Your Project

Most games involve build multiple decks. Initially, you might think to put all of your Ruby code inside one file. That can work, but it gets slow and cumbersome. Instead, I like to organize my code into separate source code files inside of a *src* directory.

Keeping your artwork in its own folder will also make it easier for you to find what you need later on. Also, using *img_dir* parameter in the *config.yml* will let you switch the entire image directory over in one

Using a Rakefile

- Setting up rake tasks
- Advanced project layout: splitting out decks into different files
- Testing individual files (build groups, ranges, id-lookup)
- Marketing images (using output as images, dependency in Rakefile)
- Rulebook figures (build groups, annotate after saving)
- Switch from built-in layouts to your own layout
- Launch what you need with Launchy

- Auto-building with Guard
- Maintaining color and black-and-white builds (build groups, multiple layout files). Changing build sessions within Guard
- Configuring different things for each build
- Git (save to json, tagging, rolling back, Gemfile.lock)

The Squib Way pt 4: Ruby Power!

Warning: To be written.

This part is about cataloging some powerful things you can do if you're willing to write some Ruby.

- Modifying XML at runtime (e.g. convert to black-and-white from color)
- Using Travis to build and then post to something like Dropbox
- Scaling the size of text based on its contents
- Advanced Array techniques: inject, transpose, map, join (use the pre-req example)
- Building newlines yourself (i.e. with your own placeholder like "%n" in Your Last Heist)
- Summarization card backs for Your Last Heist as an example
- "Lacks" string for Your Last Heist
- Rules doc written in Markdown

Squib in Action

Squib is in use by a lot of people! You can learn a lot from looking at how a whole project is put together.

A good way to peruse Squib code is to search for Ruby files on GitHub that have the phrase `require 'squib'` in them. And these are the just the people who have decided to release their code open source!

My Projects

Here are some of my own board and card games that use Squib. They are all under "active" development, which means that sometimes I leave them alone for long periods of time ;)

- Your Last Heist is a big-box cooperative game. Lots of really cool Squib things in here, including lots of Rake features, color+bw, and showing how skills can "level up" on their backs by diff'ing the stats in Squib. You can see what the components look like at the game website. Also: the best game I've ever made.
- Victory Point Salad. A card-only game with lots and lots and lots of cards. Pretty straightforward as far as Squib usage goes, but it's a good peek into how I like to use Squib. Also: the funniest game I've made.
- Junk Land A game I made prior to making starting Squib, but then ported over to Squib while developing Squib. Uses some strange features of SVG, but also a good intro. Also: the scrappiest game I've made.

Note: Want to donate back to Squib? Volunteer to playtest these games :)

Open Source Projects using Squib

Poking around GitHub, here are a few sightings of Squib:

- Ecovalia - a game rapidly prototyped in a hackathon. Squib is featured in their video!
- Werewolf implemented and even uses GitHub releases!
- Cult Following is a neat-looking project
- Mad World uses CircleCI to build, even with some custom fonts.

Other Projects Using Squib

Here are some closed-source projects that use Squib:

- ScrapyardArmory's Dysplacement was used with Squib, and **won** the Worker Placement contest over at TheGameCrafter.
- One Last Job - A 2-player Assymetrical Heist Card Game (from regular Squib contributor Brian Cronin)

Want yours here?

Create an issue on Github and ask for a link her and we'll add it here!

Squib + Game-Icons.net

I believe that, in prototyping, you want to focus on getting your idea to the table as fast as possible. Artwork is the kind of thing that can wait for later iterations once you know your game is a good one.

But! Playtesting with just text is a real drag.

Fortunately, there's this amazing project going on over at http://game-icons.net. They are in the process of building a massive library of gaming-related icons.

As a sister project to Squib, I've made a Ruby gem that interfaces with the Game Icons library. With this gem, you can access Game Icons files, and even manipulate them as they go into your game.

Here are some instructions for working with the Game Icons gem into Squib.

Install the Gem

To get the gem, do:

$ gem install game_icons

The library update frequently, so it's a good idea to upgrade whenever you can.

$ gem up game_icons

If you are using Bundler, go ahead and put this in your Gemfile:

```
gem 'game_icons'
```

And then run `bundle install` to install it from there.

The `game_icons` gem has no required dependencies. However, if you want to manipulate the SVG

To begin using the gem, just require it:

```
require 'game_icons'
```

Find Your Icon

Game-Icons.net has a search engine with some great tagging. Find the icon that you need. The gem will need the "name" of your icon. You can get this easily from the URL. For example:

> http://game-icons.net/lorc/originals/meat.html

could be called:

```
'meat'
:meat
```

Symbols are okay too (really, anything that responds to `to_s` will suffice). Spaces are replaced with a dash:

```
'police-badge'
:police_badge
```

However, some icons have the same name but different authors. To differentiate these, you put the author name before a slash. Like this:

```
'lorc/meat'
'andymeneely/police-badge'
```

To get the Icon, you use `GameIcons#get`:

```
GameIcons.get(:meat)
GameIcons.get('lorc/meat')
GameIcons.get(:police_badge)
GameIcons.get('police-badge')
GameIcons.get('andymeneely/police-badge')
```

If you want to know all the icon names, you can always use:

```
GameIcons.names # returns the list of icon names
```

If you end up misspelling one, the gem will suggest one:

```
irb(main):005:0> GameIcons.get(:police_badg)
RuntimeError: game_icons: could not find icon 'police_badg'. Did you mean any of these? police-badge
```

Use the SVG File

If you just want to use the icon in your game, you can just use the `file` method:

```
svg file: GameIcons.get(:police-badge).file
```

Recolor the SVG file

The gem will also allow you to recolor the icon as you wish, setting foreground and background:

```
# recolor foreground and background to different shades of gray
svg data: GameIcons.get('glass-heart').
                    recolor(fg: '333', bg: 'ccc').
                    string

# recolor with opacity
svg data: GameIcons.get('glass-heart').
                    recolor(fg: '333', bg: 'ccc',
                            fg_opacity: 0.25, bg_opacity: 0.75).
                    string
```

Use the SVG XML Data

SVGs are just XML files, and can be manipulated in their own clever ways. GameIcons is super-consistent in the way they format their SVGs - the entire icon is flattened into one path. So you can manipulate how the icon looks in your own way. Here's an example of using straight string substitution:

```
svg data: GameIcons.get(:meat).string.gsub('fill="#fff"', 'fill="#abc"')
```

Here's a fun one. It replaces all non-white colors in your SVG with black through the SVG:

```
svg data: GameIcons.get(:meat).string.gsub(':#ffffff', 'snarfblat').
                                      gsub(/:#[0-9a-f]{6}/, ':#000000').
                                      gsub('snarfblat', ':#ffffff')
```

XML can also be manipulated via CSS or XPATH queries via the `nokogiri` library, which Squib has as a dependency anyway. Like this:

```
doc = Nokogiri::XML(GameIcons.get(:meat).string)
doc.css('path')[1]['fill'] = #f00 # set foreground color to red
svg data: doc.to_xml
```

Path Weirdness

Inkscape and Squib's libRSVG renderer can lead to unexpected results for some icons. This has to do with a discrepancy in how path data is interpreted according to the specification. (Specifically, negative numbers need to have a space before them in the path data.) The fix for this is quick and easy, and the gem can do this for you:

```
GameIcons.get(:sheep).correct_pathdata.string # corrects path data
```

Squib + Git

> **Warning:** To be written

Ideas:

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

Autobuild with Guard

> **Warning:** Under construction - going to fold this into the Workflow guide. For now, you can see my samples. This is mostly just a brain dump.

Throughout your prototyping process, you'll be making small adjustments and then examining the graphical output. Re-running your code can get tedious, because you're cognitively switching from one context to another, e.g. editing x-y coordinates to running your code.

Ruby has a great tool for this: Guard. With Guard, you specify one configuration file (i.e. a `Guardfile`), and then Guard will *watch* a set of files, then execute a *task*. There are many ways of executing a task, but the cleanest way is via a `rake` in a `Rakefile`.

Project layout

Here's our project layout:

```
.
- config.yml
- Gemfile
- Guardfile
- img
|   - robot-golem.svg
- layouts
|   - characters.yml
|   - skills.yml
- Rakefile
- src
  - characters.rb
  - skills.rb
```

Using Guard + Rake

Guard is a gem, just like Squib. When using Guard, I recommend also using Bundler. So your Gemfile will look like this.

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

gem 'squib'      # the most important part!
gem 'guard'      # guard is what watches
gem 'guard-rake' # this hooks Guard into Rake
gem 'wdm', '>= 0.1.0' if Gem.win_platform? # optional, for watching directories
```

And then your Rakefile might look something like this

```ruby
# This is a sample Rakefile
require 'squib'

desc 'Build all decks black-and-white'
task default: [:characters, :skills]

desc 'Build all decks with color'
task color: [:with_color, :default]

desc 'Enable color build'
task :with_color do
  puts "Enabling color build"
  Squib.configure img_dir: 'color'
end

desc 'Build the character deck'
task :characters do
  puts "Building characters"
  load 'src/characters.rb'
end

desc 'Build the skills deck'
task :skills do
  puts "Building skills"
  load 'src/skills.rb'
end
```

To get our images directory set, and to turn on proress bars (which I recommend when working under Guard), you'll need a `config.yml` file that looks like this.

```yaml
img_dir: bw # is overridden by Rakefile, but in case we dont specify
progress_bars: true
```

Note that we are using `load` instead of `require` to run our code. In Ruby, `require` will only run our code once, because it's about loading a library. The `load` will run Ruby code no matter whether or not it's been loaded before. This doesn't usually matter, unless you're running under Guard.

And then our Guardfile

```ruby
group :characters do
  guard 'rake', task: :characters do # when triggered, do "rake characters"
    watch %r{src/characters.rb}  # a regular expression that matches our file
    watch %r{img/.*}             # watch every file inside of our img directory
    watch %r{.*\.xlsx$}          # Any excel file, anywhere
    watch %r{.*\.yml}            # Any yml file, anywhere (config or layout)
  end
end

group :skills do
  guard 'rake', task: :skills do # when triggered, do "rake skills"
    watch %r{src/skills.rb} # a regular expression that matches our file
    watch %r{img/.*}        # watch every file inside of our img directory
    watch %r{.*\.xlsx$}     # Any excel file, anywhere
    watch %r{.*\.yml}       # Any yml file, anywhere (config or layout)
  end
end
```

So, let's say we're working on our Character deck. To run all this we can kick off our Guard with:

```
$ bundle exec guard -g characters
14:45:20 - INFO - Run 'gem install win32console' to use color on Windows
14:45:21 - INFO - Starting guard-rake characters
14:45:21 - INFO - running characters
Loading SVG(s) <=============================================> 100% Time: 00:00:00
Saving PNGs to _output/character__* <=======================> 100% Time: 00:00:00
]2;[running rake task: characters] watched files: []
[1] Characters guard(main)> ow watching at 'C:/Users/andy/code/squib/samples/project'
```

Guard will do an initial build, then wait for file changes to be made. From here, once we edit and save anything related to characters - any Excel file, our `characters.rb` file, any YML file, etc, we'll rebuild our images.

Guard can do much, much more. It opens up a debugging console based on pry, which means if your code is broken you can test things out right there.

Guard also supports all kinds of notifications too. By default it tends to beep, but you can also have visual bells and other notifications.

To quit guard, type `quit` on their console. Or, you can do `Ctrl+C` to quit.

Enjoy!

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. 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 or layout rules.

You must use *named parameters* rather than *positional parameters*. For example:

```
save(:png) # wrong
```

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 arguments (2 for 0..1) (ArgumentError)
    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) # the right way
```

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 # narf has no effect
```

This can be troublesome when you accidentally misspell an option and don't realize it.

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.

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
```

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

Examples

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

```
require 'squib'

data = { 'name' => ['Thief', 'Grifter', 'Mastermind'],
        'type' => ['Thug', 'Thinker', 'Thinker'],
        'level' => [1, 2, 3] }

Squib::Deck.new(width: 825, height: 1125, cards: 3) do
  # Default range is :all
  background color: :white
  text str: data['name'], x: 250, y: 55, font: 'Arial 54'
  text str: data['level'], x: 65, y: 40, font: 'Arial 72'

  # Could be explicit about using :all, too
  text range: :all,
       str: data['type'], x: 40, y: 128, font: 'Arial 18',
```

```
      width: 100, align: :center

  # Ranges are inclusive, zero-based
  text range: 0..1, str: 'Thief and Grifter only!!', x: 25, y:200

  # Integers are also allowed
  text range: 0, str: 'Thief only!', x: 25, y: 250

  # Negatives go from the back of the deck
  text range: -1, str: 'Mastermind only!', x: 25, y: 250
  text range: -2..-1, str: 'Grifter and Mastermind only!', x: 25, y: 650

  # We can use Arrays too!
  text range: [0, 2], str: 'Thief and Mastermind only!!', x: 25, y:300

  # Just about everything in Squib can be given an array that
  # corresponds to the deck's cards. This allows for each card to be styled differently
  # This renders three cards, with three strings that had three different colors at three different locations.
  text str: %w(red green blue),
       color: [:red, :green, :blue],
       x: [40, 80, 120],
       y: [700, 750, 800]

  # Useful idiom: construct a hash from card names back to its index (ID),
  # then use a range. No need to memorize IDs, and you can add cards easily
  id = {} ; data['name'].each_with_index{ |name, i| id[name] = i}
  text range: id['Thief']..id['Grifter'],
       str: 'Thief through Grifter with id lookup!!',
       x:25, y: 400

  # Useful idiom: generate arrays from a column called 'type'
  type = {}; data['type'].each_with_index{ |t, i| (type[t] ||= []) << i}
  text range: type['Thinker'],
       str: 'Only for Thinkers!',
       x:25, y: 500

  # Useful idiom: draw a different number of images for different cards
  hearts = [nil, 1, 2] # i.e. card 0 has no hearts, card 2 has 2 hearts drawn
  1.upto(2).each do |n|
    range = hearts.each_index.select { |i| hearts[i] == n}
    n.times do |i|
      svg file: 'glass-heart.svg', range: range,
          x: 150, y: 55 + i * 42, width: 40, height: 40
    end
  end

  rect color: 'black' # just a border
  save_sheet prefix: 'ranges_', columns: 3
end
```

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:

```ruby
# 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:

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

Then your script looks like this:

```ruby
# 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.

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*.

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).

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
```

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
```

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

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

fantasy.yml

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

economy.yml

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

tuck_box.yml

Based on TheGameCrafter's template.

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

hand.yml

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

playing_card.yml

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

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.

Squib::DataFrame, or a Hash of Arrays

In both DSL methods, Squib will return a "data frame" (literally of type `Squib::DataFrame`). The best way to think of this is a `Hash` of `Arrays`, where each column is a key in the hash, and every element of each Array represents a data point on a card.

The data import methods expect you to structure your Excel sheet or CSV 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 on how the data can be imported.

The `Squib::DataFrame` class provides much more than what a `Hash` provides, however. The *Squib::DataFrame*

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*).

Here are some examples, which lives here

```
require 'squib'

Squib::Deck.new(width: '1.5in', height: '1.5in') do
  background color: :white

  # We can use our DSL-method to use inches
  # Computed using @dpi (set to 300 by default)
  bleed = inches(0.125)
  cut   = inches(1.25)
  rect x: bleed, y: bleed,
       width: cut, height: cut,
       dash: '0.5mm 0.5mm' # yes, units are in dashes too

  # other units too
  cm(2)              # We can also use cm this way
  cm(2) + inches(2)  # We can mix units too

  # Or we can use a string ending with cm or in
  safe_margin = '0.25 in' #you can have a space too
  safe_width  = '1 in'
  safe_height = '1.0 in  ' # trailing space is ok too
  rect x: safe_margin, y: safe_margin,
            width: safe_width, height: safe_height,
       radius: '2 mm '

  # We can also do stuff in layout. Check out the yml file...
  #  (even cleaner in Yaml since we don't need quotes!)
  use_layout file: 'using_units.yml'
```

```
  text str: 'Hello.', layout: :example

  save prefix: 'units_', format: :png
end
```

CHAPTER 8

Specifying Colors & Gradients

Colors

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'
```

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.

by custom name

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

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.

Samples

Code is maintained in the repository here in case you need some of the assets referenced.

Sample: colors and color constants

```
require 'squib'

Squib::Deck.new(width: 825, height: 1125, cards: 1) do
  background color: :white

  y = 0
  text color: '#f00', str: '3-hex', x: 50, y: y += 50
  text color: '#f00', str: '3-hex (alpha)', x: 50, y: y += 50
  text color: '#ff0000', str: '6-hex', x: 50, y: y += 50
  text color: '#ff000099', str: '8-hex(alpha)', x: 50, y: y += 50
  text color: '#ffff00000000', str: '12-hex', x: 50, y: y += 50
  text color: '#ffff000000009999', str: '12-hex (alpha)', x: 50, y: y += 50
  text color: :burnt_orange, str: 'Symbols of constants too', x: 50, y: y += 50
  text color: '(0,0)(400,0) blue@0.0 red@1.0', str: 'Linear gradients!', x: 50, y: y += 50
  text color: '(200,500,10)(200,500,100) blue@0.0 red@1.0', str: 'Radial gradients!', x: 50, y: y += 50
```

```
  # see gradients.rb sample for more on gradients

  save_png prefix: 'colors_'
end

# This script generates a table of the built-in constants
Squib::Deck.new(width: 3000, height: 1500) do
  background color: :white
  colors = (Cairo::Color.constants - %i(HEX_RE Base RGB CMYK HSV X11))
  colors.sort_by! {|c| Cairo::Color.parse(c).to_s}
  x, y, w, h = 0, 0, 300, 50
  colors.each_with_index do |color, i|
    rect x: x, y: y, width: w, height: h, fill_color: color
    text str: color.to_s, x: x + 5, y: y + 13, font: 'Sans Bold 16',
         color: (Cairo::Color.parse(color).to_hsv.v > 0.9) ? '#000' : '#fff'
    y += h
    if y > @height
      x += w
      y = 0
    end
  end
  save_png prefix: 'color_constants_'
end
```

Sample: gradients

```
require 'squib'

Squib::Deck.new do
  # Just about anywhere Squib takes in a color it can also take in a gradient too
  # The x-y coordinates on the card itself,
  # and then color stops are defined between 0 and 1
  background color: '(0,0)(0,1125) #ccc@0.0 #111@1.0'
  line stroke_color: '(0,0)(825,0) #111@1.0 #ccc@0.0',
      x1: 0, y1: 600, x2: 825, y2: 600,
       stroke_width: 15

  # Radial gradients look like this
  circle fill_color: '(425,400,2)(425,400,120) #ccc@0.0 #111@1.0',
         x: 415, y: 415, radius: 100, stroke_color: '#0000'
  triangle fill_color: '(650,400,2)(650,400,120) #ccc@0.0 #111@1.0',
           stroke_color: '#0000',
           x1: 650, y1: 360,
           x2: 550, y2: 500,
           x3: 750, y3: 500

  # Gradients are also good for beveling effects:
  rect fill_color: '(0,200)(0,600) #111@0.0 #ccc@1.0',
       x: 30, y: 350, width: 150, height: 150,
       radius: 15, stroke_color: '#0000'
  rect fill_color: '(0,200)(0,600) #111@1.0 #ccc@0.0',
       x: 40, y: 360, width: 130, height: 130,
       radius: 15, stroke_color: '#0000'

  # Alpha transparency can be used too
  text str: 'Hello, world!', x: 75, y: 700, font: 'Sans Bold 72',
```

```
      color: '(0,0)(825,0) #000f@0.0 #0000@1.0'

  save_png prefix: 'gradient_'
end
```

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.

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.

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`.

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.

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 pixels
```

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
```

Markup

See *Markup* in *text*.

Samples

These samples are maintained in the repository here in case you need some of the assets referenced.

Sample: _text.rb

```ruby
require 'squib'
require 'squib/sample_helpers'

Squib::Deck.new(width: 1000, height: 1250) do
  draw_graph_paper width, height

  sample 'Font strings are quite expressive. Specify family, modifiers, then size. Font names with spaces in them should end with a comma to help with parsing.' do |x, y|
    text font: 'Arial bold italic 32', str: 'Bold and italic!', x: x, y: y - 50
    text font: 'Arial weight=300 32', str: 'Light bold!', x: x, y: y
    text font: 'Times New Roman, 32', str: 'Times New Roman', x: x, y: y + 50
    text font: 'NoSuchFont,Arial 32', str: 'Arial Backup', x: x, y: y + 100
  end

  sample 'Specify width and height to see a text box. Also: set "hint" to see the extents of your text box' do |x, y|
    text str: 'This has fixed width and height.', x: x, y: y,
         hint: :red, width: 300, height: 100, font: 'Serif bold 24'
  end

  sample 'If you specify the width only, the text will ellipsize.' do |x, y|
    text str: 'The meaning of life is 42', x: x - 50, y: y,
         hint: :red, width: 350, font: 'Serif bold 22'
  end

  sample 'If you specify the width only, and turn off ellipsize, the height will auto-stretch.' do |x, y|
    text str: 'This has fixed width, but not fixed height.', x: x, y: y,
         hint: :red, width: 300, ellipsize: false, font: 'Serif bold 24'
  end

  sample 'The text method returns the ink extents of each card\'s rendered text. So you can custom-fit a shape around it.' do |x, y|
    ['Auto fit!', 'Auto fit!!!!' ].each.with_index do |str, i|
      text_y = y + i * 50
      extents = text str: str, x: x, y: text_y, font: 'Sans Bold 24'

      # Extents come back as an array of hashes, which can get split out like this
      text_width  = extents[0][:width]
      text_height = extents[0][:height]
      rect x: x, y: text_y, width: text_width, height: text_height, radius: 10,
           stroke_color: :purple, stroke_width: 3
    end
  end

  sample 'Text can be rotated about the upper-left corner of the text box. Unit is in radians.' do |x, y|
    text str: 'Rotated', hint: :red, x: x, y: y, angle: Math::PI / 6
  end
```

```

  save_png prefix: '_text_'
end
```

Sample: text_options.rb

```
# encoding: UTF-8
require 'squib'

data = { 'name' => ['Thief', 'Grifter', 'Mastermind'],
         'level' => [1, 2, 3] }
longtext = "This is left-justified text, with newlines.\nWhat do you know about tweetle beetles? well... When tweetle beetles fight, it's called a tweetle beetle battle. And when they battle in a puddle, it's a tweetle beetle puddle battle. AND when tweetle beetles battle with paddles in a puddle, they call it a tweetle beetle puddle paddle battle. AND... When beetles battle beetles in a puddle paddle battle and the beetle battle puddle is a puddle in a bottle... ..they call this a tweetle beetle bottle puddle paddle battle muddle."

Squib::Deck.new(width: 825, height: 1125, cards: 3) do
  background color: :white
  rect x: 15, y: 15, width: 795, height: 1095, x_radius: 50, y_radius: 50
  rect x: 30, y: 30, width: 128, height: 128, x_radius: 25, y_radius: 25

  # Arrays are rendered over each card
  text str: data['name'], x: 250, y: 55, font: 'Arial weight=900 54'
  text str: data['level'], x: 65, y: 40, font: 'Arial 72', color: :burnt_orange

  text str: 'Font strings are expressive!', x:65, y: 200,
       font: 'Impact bold italic 36'

  text str: 'Font strings are expressive!', x:65, y: 300,
       font: 'Arial,Verdana weight=900 style=oblique 36'

  text str: 'Font string sizes can be overridden per card.', x: 65, y: 350,
       font: 'Impact 36', font_size: [16, 20, 24]

  text str: 'This text has fixed width, fixed height, center-aligned, middle-valigned, and has a red hint',
       hint: :red,
       x: 65, y: 400,
       width: 300, height: 125,
       align: :center, valign: 'MIDDLE', # these can be specified with case-insenstive strings too
       font: 'Serif 16'

  extents = text str: 'Ink extent return value',
       x: 65, y: 550,
       font: 'Sans Bold', font_size: [16, 20, 24]
  margin = 10
  # Extents come back as an array of hashes, which can get split out like this
  ws = extents.inject([]) { |arr, ext| arr << ext[:width] + 10; arr }
  hs = extents.inject([]) { |arr, ext| arr << ext[:height] + 10; arr }
  rect x: 65 - margin / 2, y: 550 - margin / 2,
       width: ws, height: hs,
       radius: 10, stroke_color: :black
```

```

  # If width & height are defined and the text will overflow the box, we can ellipsize.
  text str: "Ellipsization!\nThe ultimate question of life, the universe, and everything to life and everything is 42",
       hint: :green, font: 'Arial 22',
       x: 450, y: 400,
       width: 280, height: 180,
       ellipsize: true

  # Text hints are guides for showing you how your text boxes are laid out exactly
  hint text: :cyan
  set font: 'Serif 20' # Impacts all future text calls (unless they specify differently)
  text str: 'Text hints & fonts are globally togglable!', x: 65, y: 625
  set font: :default # back to Squib-wide default
  hint text: :off
  text str: 'See? No hint here.',
        x: 565, y: 625,
        font: 'Arial 22'

  # Text can be rotated, in radians, about the upper-left corner of the text box.
  text str: 'Rotated',
        x: 565, y: 675, angle: 0.2,
        font: 'Arial 18', hint: :red

  # Text can be justified, and have newlines
  text str: longtext, font: 'Arial 16',
       x: 65, y: 700,
       width: '1.5in', height: inches(1),
       justify: true, spacing: -6

  # Here's how you embed images into text.
  # Pass a block to the method call and use the given context
  embed_text = 'Embedded icons! Take 1 :tool: and gain 2:health:. If Level 2, take 2 :tool:'
  text(str: embed_text, font: 'Sans 18',
       x: '1.8in', y: '2.5in', width: '0.85in',
       align: :left, ellipsize: false) do |embed|
    embed.svg key: ':tool:',   width: 28, height: 28, file: 'spanner.svg'
    embed.svg key: ':health:', width: 28, height: 28, file: 'glass-heart.svg'
  end

  text str: 'Fill n <span fgcolor="#ff0000">stroke</span>',
       color: :green, stroke_width: 2.0, stroke_color: :blue,
       x: '1.8in', y: '2.9in', width: '0.85in', font: 'Sans Bold 26', markup: true

  text str: 'Stroke n <span fgcolor="#ff0000">fill</span>',
       color: :green, stroke_width: 2.0, stroke_color: :blue, stroke_strategy: :stroke_first,
       x: '1.8in', y: '3.0in', width: '0.85in', font: 'Sans Bold 26', markup: true

  text str: 'Dotted',
       color: :white, stroke_width: 2.0, dash: '4 2', stroke_color: :black,
       x: '1.8in', y: '3.1in', width: '0.85in', font: 'Sans Bold 26', markup: true
  #
  text str: "<b>Markup</b> is <i>quite</i> <s>'easy'</s> <span fgcolor=\"\#ff0000\">awesome</span>. Can't beat those \"smart\" 'quotes', now with 10--20% more en-dashes --- and em-dashes --- with explicit ellipses too...",
```

```
        markup: true,
        x: 50, y: 1000,
        width: 750, height: 100,
        valign: :bottom,
        font: 'Serif 18', hint: :cyan

  save prefix: 'text_options_', format: :png
end
```

Sample: embed_text.rb

```
require 'squib'

Squib::Deck.new do
  background color: :white
  rect x: 0, y: 0, width: 825, height: 1125, stroke_width: 2.0

  embed_text = 'Take 11 :tool: and gain 2 :health:. Take <b>2</b> :tool: <i>and gain 3 :purse: if level 2.</i>'
  text(str: embed_text, font: 'Sans 21',
       x: 0, y: 0, width: 180, hint: :red,
       align: :left, ellipsize: false, justify: false) do |embed|
    embed.svg key: ':tool:',   width: 28, height: 28, file: 'spanner.svg'
    embed.svg key: ':health:', width: 28, height: 28, file: 'glass-heart.svg'
    embed.png key: ':purse:',  width: 28, height: 28, file: 'shiny-purse.png'
  end

  embed_text = 'Middle align: Take 1 :tool: and gain 2 :health:. Take 2 :tool: and gain 3 :purse:'
  text(str: embed_text, font: 'Sans 21',
       x: 200, y: 0, width: 180, height: 300, valign: :middle,
       align: :left, ellipsize: false, justify: false, hint: :cyan) do |embed|
    embed.svg key: ':tool:',   width: 28, height: 28, file: 'spanner.svg'
    embed.svg key: ':health:', width: 28, height: 28, file: 'glass-heart.svg'
    embed.png key: ':purse:',  width: 28, height: 28, file: 'shiny-purse.png'
  end

  embed_text = 'This :tool: aligns on the bottom properly. :purse:'
  text(str: embed_text, font: 'Sans 21',
       x: 400, y: 0, width: 180, height: 300, valign: :bottom,
       align: :left, ellipsize: false, justify: false, hint: :green) do |embed|
    embed.svg key: ':tool:',   width: 28, height: 28, file: 'spanner.svg'
    embed.svg key: ':health:', width: 28, height: 28, file: 'glass-heart.svg'
    embed.png key: ':purse:',  width: 28, height: 28, file: 'shiny-purse.png'
  end

  embed_text = 'Yes, this wraps strangely. We are trying to determine the cause. These are 1 :tool::tool::tool: and these are multiple :tool::tool: :tool::tool:'
  text(str: embed_text, font: 'Sans 18',
       x: 600, y: 0, width: 180, height: 300, wrap: :word_char,
       align: :left, ellipsize: false, justify: false, hint: :cyan) do |embed|
    embed.svg key: ':tool:', width: 28, height: 28, file: 'spanner.svg'
  end

  embed_text = ':tool:Justify will :tool: work too, and :purse: with more words just for fun'
```

```
text(str: embed_text, font: 'Sans 21',
     x: 0, y: 320, width: 180, height: 300, valign: :bottom,
     align: :left, ellipsize: false, justify: true, hint: :magenta) do |embed|
  embed.svg key: ':tool:',   width: 28, height: 28, file: 'spanner.svg'
  embed.svg key: ':health:', width: 28, height: 28, file: 'glass-heart.svg'
  embed.png key: ':purse:',  width: 28, height: 28, file: 'shiny-purse.png'
end

embed_text = 'Right-aligned works :tool: with :health: and :purse:'
text(str: embed_text, font: 'Sans 21',
     x: 200, y: 320, width: 180, height: 300, valign: :bottom,
     align: :right, ellipsize: false, justify: false, hint: :magenta) do |embed|
  embed.svg key: ':tool:',   width: 28, height: 28, file: 'spanner.svg'
  embed.svg key: ':health:', width: 28, height: 28, file: 'glass-heart.svg'
  embed.png key: ':purse:',  width: 28, height: 28, file: 'shiny-purse.png'
end

embed_text = ':tool:Center-aligned works :tool: with :health: and :purse:'
text(str: embed_text, font: 'Sans 21',
     x: 400, y: 320, width: 180, height: 300,
     align: :center, ellipsize: false, justify: false, hint: :magenta) do |embed|
  embed.svg key: ':tool:',   width: 28, height: 28, data: File.read('spanner.svg')
  embed.svg key: ':health:', width: 28, height: 28, file: 'glass-heart.svg'
  embed.png key: ':purse:',  width: 28, height: 28, file: 'shiny-purse.png'
end

embed_text = 'Markup --- and typography replacements --- with ":tool:" icons <i>won\
↪'t</i> fail'
text(str: embed_text, font: 'Serif 18', markup: true,
     x: 600, y: 320, width: 180, height: 300,
     align: :center, hint: :magenta) do |embed|
  embed.svg key: ':tool:',   width: 28, height: 28, file: 'spanner.svg'
end

embed_text = ':tool:' # JUST the icon
text(str: embed_text, x: 0, y: 640, width: 180, height: 50, markup: true,
   font: 'Arial 21', align: :center, valign: :middle, hint: :red) do |embed|
  embed.svg key: ':tool:', width: 28, height: 28, file: 'spanner.svg'
end

embed_text = ':purse:' # JUST the icon
text(str: embed_text, x: 200, y: 640, width: 180, height: 50, markup: true,
   font: 'Arial 21', align: :center, valign: :middle, hint: :red) do |embed|
  embed.png key: ':purse:', width: 28, height: 28, file: 'shiny-purse.png'
end

embed_text = ":tool: Death to Nemesis bug 103!! :purse:"
text(str: embed_text, font: 'Sans Bold 24', stroke_width: 2,
     color: :red, stroke_color: :blue, dash: '3 3', align: :left,
     valign: :middle, x: 0, y: 700, width: 380, height: 150,
     hint: :magenta) do |embed|
  embed.svg key: ':tool:', file: 'spanner.svg', width: 32, height: 32
  embed.png key: ':purse:', file: 'shiny-purse.png', width: 32, height: 32
end

embed_text = 'You can adjust the icon with dx and dy. Normal: :tool: Adjusted:␣
↪:heart:'
text(str: embed_text, font: 'Sans 18', x: 400, y: 640, width: 180,
```

```
       height: 300, hint: :magenta) do |embed|
    embed.svg key: ':tool:', width: 28, height: 28, file: 'spanner.svg'
    embed.svg key: ':heart:', width: 28, height: 28, dx: 10, dy: 10,
              file: 'glass-heart.svg'
  end

  embed_text = "Native sizes work too\n:tool:\n\n\n\n\n\n:shiny-purse:\n\n\n\n\n\n:tool2:"
  text(str: embed_text, font: 'Sans 18', x: 600, y: 640, width: 180,
       height: 475, hint: :magenta) do |embed|
    embed.svg key: ':tool:', width: :native, height: :native,
              file: 'spanner.svg'
    embed.svg key: ':tool2:', width: :native, height: :native,
              data: File.open('spanner.svg','r').read
    embed.png key: ':shiny-purse:', width: :native, height: :native,
              file: 'shiny-purse.png'
  end

  save_png prefix: 'embed_'
end

Squib::Deck.new(cards: 3) do
  background color: :white
  str = 'Take 1 :tool: and gain 2 :health:.'
  text(str: str, font: 'Sans', font_size: [18, 26, 35],
       x: 0, y: 0, width: 180, height: 300, valign: :bottom,
       align: :left, ellipsize: false, justify: false, hint: :cyan) do |embed|
    embed.svg key: ':tool:',   width: [28, 42, 56], height: [28, 42, 56], file: 'spanner.svg'
    embed.svg key: ':health:', width: [28, 42, 56], height: [28, 42, 56], file: 'glass-heart.svg'
  end
  save_sheet prefix: 'embed_multisheet_', columns: 3
end
```

Sample: config_text_markup.rb

```
require 'squib'

Squib::Deck.new(config: 'config_text_markup.yml') do
  background color: :white
  text str: %{"'Yaml ain't markup', he says"},
       x: 10, y: 10, width: 300, height: 200, font: 'Serif 20',
       markup: true, hint: :cyan

  text str: 'Notice also the antialiasing method.',
       x: 320, y: 10, width: 300, height: 200, font: 'Arial Bold 20'

  save_png prefix: 'config_text_'
end

Squib::Deck.new(config: 'config_disable_quotes.yml') do
  text str: %{This has typographic sugar --- and ``explicit'' quotes --- but the quotes are "dumb"},
       x: 10, y: 10, width: 300, height: 200, font: 'Serif 20',
       markup: true, hint: :cyan
```

```
  save_png prefix: 'config_disable_text_'
end
```

```
# We can configure what characters actually get replaced by quoting them with unicode
↪code points.
lsquote: "\u2018" #note that Yaml wants double quotes here to use escape chars
rsquote: "\u2019"
ldquote: "\u201C"
rdquote: "\u201D"
em_dash: "\u2014"
en_dash: "\u2013"
ellipsis: "\u2026"
antialias: gray
```

```
# If we want to disable smart quoting and only allow explicit quoting within markup,
# use this option
smart_quotes: false
```

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:

```yaml
# 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.

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
```

These are read-only - you will not be able to change these.

Set options programmatically

You can also use *Squib.configure* to override anything in the config file. Use it like this:

```ruby
# This is a sample Rakefile
require 'squib'

desc 'Build all decks black-and-white'
task default: [:characters, :skills]

desc 'Build all decks with color'
task color: [:with_color, :default]

desc 'Enable color build'
task :with_color do
  puts "Enabling color build"
  Squib.configure img_dir: 'color'
end

desc 'Build the character deck'
task :characters do
  puts "Building characters"
  load 'src/characters.rb'
end

desc 'Build the skills deck'
task :skills do
  puts "Building skills"
  load 'src/skills.rb'
end
```

See *The Squib Way pt 3: Workflows* for how we put this to good use.

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:

```ruby
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:

```ruby
require 'squib'

Squib::Deck.new(width: 75, height: 75, cards: 2) do
  # puts "Groups enabled by environment: #{groups.to_a}"

  text str: ['A', 'B']

  build :print_n_play do
    rect
    save_sheet prefix: 'build_groups_bw_'
  end

  build :color do
    rect stroke_color: :red, dash: '5 5'
    save_png prefix: 'build_groups_color_'
  end

  build :test do
    save_png range: 0, prefix: 'build_groups_'
  end

end

# Here's how you can run this on the command line:
#
```

```
# --- OSX/Linux (bash or similar shells) ---
# $ ruby build_groups.rb
# $ SQUIB_BUILD=color ruby build_groups.rb
# $ SQUIB_BUILD=print_n_play,test ruby build_groups.rb
#
# --- Windows CMD ---
# $ ruby build_groups.rb
# $ set SQUIB_BUILD=color && ruby build_groups.rb
# $ set SQUIB_BUILD=print_n_play,test && ruby build_groups.rb
#
# Or, better yet... use a Rakefile like the one provided in this gist!
```

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

Groups can be enabled and disabled in several ways:

- The *enable_build* and *disable_build* DSL methods within a given `Squib::Deck` 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. This is handy for enabling builds on the command line (e.g. turn on color, work in that for a while, then turn it off)
- Furthermore, you can use *Squib.enable_build_globally* and *Squib.disable_build_globally* to manipulate `SQUIB_BUILD` environment variable programmatically (e.g. from a Rakefile, inside a Guard session, or other build script).

The *The Squib Way pt 3: Workflows* tutorial covers how to work these features into your workflow.

Note: There should be no need to set the SQUIB_BUILD variable globally on your system (e.g. at boot). The intent is to set SQUIB_BUILD as part of your session.

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:

```
# Example Rakefile that makes use of build groups

desc 'Build black-and-white by default'
task default: [:bw]

desc 'Build both bw and color'
task both: [:bw, :color]

desc 'Build black-and-white only'
task :bw do
  Squib.enable_build_globally :print_n_play
  load 'build_groups.rb'
end

desc 'Build the color version only'
task :color do
  Squib.enable_build_globally :color
  load 'build_groups.rb'
end

```

```ruby
desc 'Build a test card'
task :color do
  Squib.enable_build_globally :test
  load 'build_groups.rb'
end
```

Thus, you can just run this code on the command line like these:

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

CHAPTER 14

Get Help and Give Help

Show Your Pride

We would also love to hear about the games you make with Squib!

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.

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.

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!

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.

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

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.

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.

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

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`.

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.

Examples

background

Fills the background with the given color

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*.

Examples

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.

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.

Examples

Use group to organize your Squib code into build groups:

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

build_groups

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

Arguments

(none)

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
```

circle

Draw a partial or complete circle centered at the given coordinates

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*.

arc_start default: `0`

angle on the circle to start an arc

arc_end default: `2 * Math::PI`

angle on the circle to end an arc

arc_direction default: `:clockwise`

draw the arc clockwise or counterclockwise from arc_start to arc_end

arc_close default: `false`

draw a straight line closing the arc

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*.

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'

```

```ruby
  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: 450, y: 600, radius: 75,
         fill_color: :gray, stroke_color: :green, stroke_width: 8.0

  circle x: 600, y: 600, radius: 75, # partial circle
         arc_start: 1, arc_end: 4, arc_direction: :counter_clockwise,
         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

  curve x1: 50,  y1: 850, cx1: 150, cy1: 700,
        x2: 625, y2: 900, cx2: 150, cy2: 700,
        stroke_width: 12.0, stroke_color: :cyan,
        fill_color: :burgundy, cap: 'round'

  ellipse x: 50, y: 925, width: 200, height: 100,
          stroke_width: 5.0, stroke_color: :cyan,
          fill_color: :burgundy

  star x: 300, y: 1000, n: 5, inner_radius: 15, outer_radius: 40,
       fill_color: :cyan, stroke_color: :burgundy, stroke_width: 5

  # default draw is fill-then-stroke. Can be changed to stroke-then-fill
  star x: 375, y: 1000, n: 5, inner_radius: 15, outer_radius: 40,
       fill_color: :cyan, stroke_color: :burgundy,
       stroke_width: 5, stroke_strategy: :stroke_first

  polygon x: 500, y: 1000, n: 5, radius: 25, angle: Math::PI / 2,
          fill_color: :cyan, stroke_color: :burgundy, stroke_width: 2

  save_png prefix: 'shape_'
end
```

cm

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

Parameters

n the number of centimeters

Examples

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

Squib.configure

Prior to the construction of a Squib::Deck, set a global default that overrides what is specified *config.yml*.

This is intended to be done prior to Squib::Deck.new, and is intended to be used inside of a Rakefile

Options

All options that are specified in *Configuration Options*

Exmaples

```
# This is a sample Rakefile
require 'squib'

desc 'Build all decks black-and-white'
task default: [:characters, :skills]

desc 'Build all decks with color'
task color: [:with_color, :default]

desc 'Enable color build'
task :with_color do
  puts "Enabling color build"
  Squib.configure img_dir: 'color'
end

desc 'Build the character deck'
task :characters do
  puts "Building characters"
  load 'src/characters.rb'
end

desc 'Build the skills deck'
task :skills do
  puts "Building skills"
  load 'src/skills.rb'
end
```

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
```

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.

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
```

Examples

To get the sample Excel files, go to its source

```
require 'squib'

Squib::Deck.new(cards: 2) do
  background color: :white

  # Outputs a hash of arrays with the header names as keys
  data = csv file: 'sample.csv'
  text str: data['Type'], x: 250, y: 55, font: 'Arial 54'
  text str: data['Level'], x: 65, y: 65, font: 'Arial 72'

  save format: :png, prefix: 'sample_csv_'

  # You can also specify the sheet, starting at 0
  data = xlsx file: 'sample.xlsx', sheet: 2
end

# CSV is also a Squib-module-level function, so this also works:
data      = Squib.csv file: 'quantity_explosion.csv' # 2 rows...
num_cards = data['Name'].size                        #          ...but 4 cards!

Squib::Deck.new(cards: num_cards) do
  background color: :white
  rect # card border
  text str: data['Name'], font: 'Arial 54'
  save_sheet prefix: 'sample_csv_qty_', columns: 4
end

# Additionally, CSV supports inline data specifically
data = Squib.csv data: <<-EOCSV
Name,Cost
Knight,3
Orc,1
EOCSV
```

Here's the sample.csv

```
Type,"Level"
Thief,1
Mastermind,2
```

Here's the quantity_explosion.csv

```
Name,Qty
Basilisk,3
High Templar,1
```

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.

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*.

Examples

Squib::DataFrame

As described in *Be Data-Driven with XLSX and CSV*, the `Squib::DataFrame` is what is returned by Squib's data import methods (*csv* and *xlsx*).

It behaves like a `Hash` of `Arrays`, so acessing an individual column can be done via the square brackets, e.g. `data['title']`.

Here are some other convenience methods in `Squib::DataFrame`

columns become methods

Through magic of Ruby metaprogramming, every column also becomes a method on the data frame. So these two are equivalent:

```
irb(main):002:0> data = Squib.csv file: 'basic.csv'
=> #<Squib::DataFrame:0x00000003764550 @hash={"h1"=>[1, 3], "h2"=>[2, 4]}>
irb(main):003:0> data.h1
=> [1, 3]
irb(main):004:0> data['h1']
=> [1, 3]
```

#columns

Returns an array of the column names in the data frame

#ncolumns

Returns the number of columns in the data frame

#col?(name)

Returns `true` if there is column `name`.

#row(i)

Returns a hash of values across all columns in the i-th row of the dataframe. Represents a single card.

#nrows

Returns the number of rows the data frame has, computed by the maximum length of any column array.

#to_json

Returns a `json` representation of the entire data frame.

#to_pretty_json

Returns a `json` representation of the entire data frame, formatted with indentation for human viewing.

#to_pretty_text

Returns a textual representation of the dataframe that emulates what the information looks like on an individual card. Here's an example:

```
             ------------------------------------
       Name | Mage                                |
       Cost | 1                                   |
Description | You may cast 1 spell per turn       |
      Snark | Magic, dude.                        |
             ------------------------------------
             ------------------------------------
       Name | Rogue                               |
       Cost | 2                                   |
Description | You always take the first turn.     |
      Snark | I like to be sneaky                 |
             ------------------------------------
             ------------------------------------
       Name | Warrior                             |
       Cost | 3                                   |
Description |
      Snark | I have a long story to tell to tes |
            | t the word-wrapping ability of pre |
            | tty text formatting.                |
             ------------------------------------
```

disable_build

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

Required Arguments

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

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
```

disable_build_globally

Disable the given build group for all future `Squib::Deck` runs.

Essentially a convenience method for setting the `SQUIB_BUILD` environment variable. See *Group Your Builds* for ways to use this effectively.

This is a member of the `Squib` module, so you must run it like this:

```
Squib.disable_build_globally :pdf
```

The intended purpose of this method is to be able to alter the environment from other build scripts, such as a Rakefile.

Required Arguments

build_group_name

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

Examples

Can be used to disable a group, overriding setting the environment variable at the command line:

```
Squib.enable_build_globally :pdf
Squib.disable_build_globally :pdf

Squib::Deck.new do
  build :pdf do
    save_pdf #does not get run regardless of incoming environment
  end
end
```

But gets overridden by an individual `Squib::Deck` programmatically enabling a build via *enable_build*:

```
Squib.enable_build_globally :pdf
Squib.disable_build_globally :pdf

Squib::Deck.new do
  enable_build :pdf
  build :pdf do
    save_pdf # this will be run no matter what
  end
end
```

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*.

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*.

Examples

enable_build

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

Required Arguments

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

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
```

disable_build_globally

Enagle the given build group for all future `Squib::Deck` runs.

Essentially a convenience method for setting the `SQUIB_BUILD` environment variable. See *Group Your Builds* for ways to use this effectively.

This is a member of the `Squib` module, so you must run it like this:

```
Squib.enable_build_globally :pdf
```

The intended purpose of this method is to be able to alter the environment from other build scripts, such as a Rakefile.

Required Arguments

build_group_name

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

Examples

Can be used to enable a group, overriding setting the environment variable at the command line:

```
Squib.enable_build_globally :pdf

Squib::Deck.new do
  build :pdf do
    save_pdf # this runs regardless of incoming environment
```

```
  end
end
```

But gets overridden by an individual `Squib::Deck` programmatically enabling a build via *enable_build*:

```
Squib.enable_build_globally :pdf

Squib::Deck.new do
  disable_build :pdf
  build :pdf do
    save_pdf # this will NOT be run no matter what
  end
end
```

grid

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

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*.

Examples

hand

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

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*

Examples

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.

Options

text default: `:off`

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

Examples

inches

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

Parameters

n the number of inches

Examples

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

line

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

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*.

Examples

mm

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

Parameters

n the number of mm

Examples

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

png

Renders PNG images.

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*.

Examples

polygon

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

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*.

Examples

rect

Draw a rounded rectangle

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

Examples

save

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

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]`

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
```

save_pdf

Lays out the cards in a gride and renders a PDF.

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*.

crop_marks default: `false`

When `true`, draws lines in the margins as guides for cutting. Crop marks factor in the `trim` (if non-zero), and can also be customized via `crop_margin_*` options (see below). Has no effect if `margin` is 0.

> **Warning:** Enabling this feature will draw lines to the edge of the page. Most PDF Readers, by default, will recognize this and scale down the entire PDF to fit in those crop marks - throwing off your overall scale. To disable this, you will need to set Print Scaling "Use original" or "None" when you go to print (this looks different for different PDF readers). Be sure to test this out before you do your big print job!!

crop_margin_bottom default: 0

The space between the bottom edge of the (potentially trimmed) card, and the crop mark. Supports *Unit Conversion*. Has no effect if `crop_marks` is `false`.

crop_margin_left default: 0

The space between the left edge of the (potentially trimmed) card, and the crop mark. Supports *Unit Conversion*. Has no effect if `crop_marks` is `false`.

crop_margin_right default: `0`

The space between the right edge of the (potentially trimmed) card, and the crop mark. Supports *Unit Conversion*. Has no effect if `crop_marks` is `false`.

crop_margin_top default: `0`

The space between the top edge of the (potentially trimmed) card, and the crop mark. Supports *Unit Conversion*. Has no effect if `crop_marks` is `false`.

crop_stroke_color default: `:black`

The color of the crop mark lines. Has no effect if `crop_marks` is `false`.

crop_stroke_dash default: `''`

Define a dash pattern for the crop marks. 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*. Has no effect if `crop_marks` is `false`.

crop_stroke_width default: `1.5`

Width of the crop mark lines. Has no effect if `crop_marks` is `false`.

Examples

save_png

Saves the given range of cards to a PNG

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.

Examples

save_sheet

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

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*.

rtl default `false`

whether to render columns right to left, used for duplex printing of card backs

Examples

showcase

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

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*

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*

Examples

This sample lives here.

```
require 'squib'

# Showcases are a neat way to show off your cards in a modern way, using a
# reflection and a persepctive effect to make them look 3D
Squib::Deck.new(cards: 4) do
  background color: '#CE534D'
  rect fill_color: '#DED4B9', x: 78, y: 78,
       width: '2.25in', height: '3.25in', radius: 32
  text str: %w(Grifter Thief Thug Kingpin),
       font: 'Helvetica,Sans weight=800 120',
       x: 78, y: 78, width: '2.25in', align: :center
  svg file: 'spanner.svg', x: (825 - 500) / 2, y: 500, width: 500, height: 500

  # Defaults are pretty sensible.
  showcase file: 'showcase.png'

  # Here's a more complete example.
  # Tons of ways to tweak it if you like - check the docs.
  showcase trim: 32, trim_radius: 32, margin: 100, face: :right,
           scale: 0.85, offset: 0.95, fill_color: :black,
           reflect_offset: 25, reflect_strength: 0.1, reflect_percent: 0.4,
           file: 'showcase2.png'

  save_png prefix: 'showcase_individual_' # to show that they're not trimmed
end
```

star

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

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*.

Examples

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.

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*.

Examples

text

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

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

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 font string docs. 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*.

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 markup 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*

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.

Examples

See *The Mighty text Method*.

triangle

Draw a triangle at the given coordinates.

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*.

Examples

use_layout

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

Options

file default: `'layout.yml'`

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

Examples

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
```

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.

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
```

Examples

To get the sample Excel files, go to its source

```
require 'squib'

Squib::Deck.new(cards: 3) do
  background color: :white

  # Reads the first sheet by default (sheet 0)
  # Outputs a hash of arrays with the header names as keys
  data = xlsx file: 'sample.xlsx'

  text str: data['Name'], x: 250, y: 55, font: 'Arial 54'
  text str: data['Level'], x: 65, y: 65, font: 'Arial 72'
  text str: data['Description'], x: 65, y: 600, font: 'Arial 36'

  save format: :png, prefix: 'sample_excel_' # save to individual pngs
end

# xlsx is also a Squib-module-level function, so this also works:
data      = Squib.xlsx file: 'explode_quantities.xlsx' # 2 rows...
num_cards = data['Name'].size                          #          ...but 4 cards!

Squib::Deck.new(cards: num_cards) do
  background color: :white
  rect # card border
  text str: data['Name'], font: 'Arial 54'
  save_sheet prefix: 'sample_xlsx_qty_', columns: 4
end


# Here's another example, a bit more realistic. Here's what's going on:
#   * We call xlsx from Squib directly - BEFORE Squib::Deck creation. This
#     allows us to infer the number of cards based on the size of the "Name"
#     field
#   * We make use of quantity explosion. Fields named "Qty" or "Quantity"
#     (any capitalization), or any other in the "qty_header" get expanded by the
#     number given
#   * We also make sure that trailing and leading whitespace is stripped
#     from each value. This is the default behavior in Squib, but the options
#     are here just to make sure.

resource_data = Squib.xlsx(file: 'sample.xlsx', sheet: 2, strip: true) 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

Squib::Deck.new(cards: resource_data['Name'].size) do
  background color: :white
  rect width: :deck, height: :deck
  text str: resource_data['Name'], align: :center, width: :deck, hint: 'red'
  text str: resource_data['Cost'], align: :right, width: :deck, hint: 'red'
  save_sheet prefix: 'sample_excel_resources_' # save to a whole sheet
end
```

yaml

Pulls deck data from a YAML files into a `Squib::DataFrame` (essentially a hash of arrays).

Parsing uses Ruby's built-in Yaml package.

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

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

The Yaml file format assumes that the entire deck is an array, then each element of the array is a hash. Every key encountered in that hash will translate to a "column" in the data frame. If a key exists in one card and not in another, then it defaults to `nil`.

> **Warning:** Case matters in your Yaml keys.

Options

file default: `'deck.yml'`

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

data default: `nil`

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

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.

Individual Pre-processing

The `yaml` 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.yaml(file: 'sample.yaml') 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
```

Examples

To get the sample Excel files, go to its source

```
require 'squib'

Squib::Deck.new(cards: 2) do
  background color: :white

  # Outputs a hash of arrays with the header names as keys
  data = csv file: 'sample.csv'
  text str: data['Type'], x: 250, y: 55, font: 'Arial 54'
  text str: data['Level'], x: 65, y: 65, font: 'Arial 72'

  save format: :png, prefix: 'sample_csv_'

  # You can also specify the sheet, starting at 0
  data = xlsx file: 'sample.xlsx', sheet: 2
end

# CSV is also a Squib-module-level function, so this also works:
data      = Squib.csv file: 'quantity_explosion.csv' # 2 rows...
num_cards = data['Name'].size                         #          ...but 4 cards!

Squib::Deck.new(cards: num_cards) do
  background color: :white
  rect # card border
  text str: data['Name'], font: 'Arial 54'
  save_sheet prefix: 'sample_csv_qty_', columns: 4
end

# Additionally, CSV supports inline data specifically
data = Squib.csv data: <<-EOCSV
Name,Cost
Knight,3
Orc,1
EOCSV
```

Here's the sample.csv

```
Type,"Level"
Thief,1
Mastermind,2
```

Here's the quantity_explosion.csv

```
Name,Qty
Basilisk,3
High Templar,1
```

CHAPTER 16

Indices and tables

- genindex
- search