

Describing Equivalence Classes in Ruby with RSpec

Dave Astels

July 19, 2007

Abstract

Ruby's Enumerable module includes a partition method that divides the collection into two parts based on the result of passing each element to the supplied block. This is handy, but sometimes there is a need to partition a collection into more than two parts. This article provides this capability. Further, it uses the exercise as an example of using RSpec.

1 Introduction

I was writing a rails app to help manage my (and my friends) collections of Magic the Gathering cards and decks. At one point I wanted to have a deck generate an HTML representation that was nicely formatted for direct pasting into blog posts. One requirement was that it not be done in some arbitrary order (or lack thereof)... it should be separated out into groups of cards of the same type: creatures, lands, sorceries, and so on.

Enumerable has the partition method, which is close:

```
enum.partition { | obj | block } => [ true_array, false_array ]
```

Returns two arrays, the first containing the elements of enum for which the block evaluates to true, the second containing the rest.

```
(1..6).partition { |i| (i&1).zero? } => [[2, 4, 6], [1, 3, 5]]
```

The problem is that it just creates 2 equivalence classes based on the true/false result of evaluating the supplied block on each element. I needed something that could generate an arbitrary number of equivalence classes.

So I pulled out RSpec and got to work.

1.1 Equivalence Classes

But first some theory.

From Wikipedia:

In mathematics, given a set X and an equivalence relation \sim on X , the equivalence class of an element a in X is the subset of all elements in X which are equivalent to a :

$$[a] = \{x \in X \mid x \sim a\}$$

The notion of equivalence classes is useful for constructing sets out of already constructed ones. The set of all equivalence classes in X given an equivalence relation \sim is usually denoted as X/\sim and called the quotient set of X by \sim .

For example, if X is the set of all cars, and \sim is the equivalence relation "has the same color as", then one particular equivalence class consists of all green cars. X/\sim could be naturally identified with the set of all car colors.

Because of the properties of an equivalence relation it holds that a is in $[a]$ and that any two equivalence classes are either equal or disjoint.

1.2 A Slight Twist

This is almost what I needed. The one difference is that equivalence classes are disjoint (i.e. each element belongs in one and only one equivalence class). I needed something a little more flexible. My driving requirement is to be able to sort cards by their types. Cards types can be somewhat hierarchical. For example, while there are distinct types (e.g. *Instant*, *Creature*, *Sorcery*), there are also *Land*, *Basic Land*, *Legendary Land*, and so on. The last two being specializations of the first.

I wanted to be able to do things like get a set of all *Legendary Lands*, and a set of all the remaining *Lands*. Notice I said **remaining**. If something goes into the *Legendary Land* pile, I don't want it in the *Land* pile as well.

So using an equivalence relation of "has the same type as" wouldn't cut it. If it looked at the core type (e.g. "Land") then specializations like "Legendary Land" wouldn't be separated out. On the other extreme, if it looked at the full type (e.g. "Legendary Land") then other specializations (e.g. "Basic Land") **would** be in their own class. And I didn't want that either... I needed something more flexible, and parameterizable.

What I needed was some form of parameterized, prioritized division into classes.

2 The First Behaviour

The first bit is often the hardest when using TDD or BDD. You want to look for the simplest case... how about the case when everything is in the same class? So the result will be a single class containing the original collection.

First we need to decide what the interface should be. I decided to stick with something similar to **partition**: use a block to define the classes. Additionally, I will pass in values that will label & help define the classes. Each element will be passed into the block with each class key. If the block results in something **true**, it means that the element belongs in that class.

Let's start by simply stating that something should be returned:

```
describe "Creating equiv classes when there is only one" do

  before(:each) do
    @equiv_classes = [0, 1, 2, 3, 4].equivalence_classes(:one) {|datum, key| true}
  end

  it "should not be nil" do
    @equiv_classes.should_not be_nil
  end
end
```

This isn't much, but it will get us started. Notice that the code in the before block is defining the interface to our new code: the name of the method, arguments, and the fact that it should take a block. The example describes the result.

Here's the skeleton of our solution:

```
module Enumerable

  def equivalence_classes(*keys, &block)
    0
  end

end

it "should behave like an indexed collection" do
  @equiv_classes.should respond_to(:[], :size, :entries)
end
```

This forces us to make another design decision, and evolve our code little bit more. So we need a collection... make it the simplest to get things green again... an Array will do:

```
def equivalence_classes(*keys, &block)
  []
end
```

Our result is a collection, but we want to go a bit further, it should be a hash so that each class can be indexed by it's key (e.g. *"Land"*, *"Creature"*, etc.). We can state that by describing what happens if we user a non-numeric index, specifically that it should be ok:

```
it "should be indexable by a symbol (since that's what we expect)" do
  lambda {@equiv_classes[:one]}.should_not raise_error
end
```

This requires a simple change:

```
def equivalence_classes(*keys, &block)
  {}
end
```

Let's say something else about the result. In this case it should contain a single class. So let's give an example that says that:

```
it "should have one element" do
  @equiv_classes.should have(1).element
end
```

Not much to do to get that working:

```
def equivalence_classes(*keys, &block)
  {:a => 1}
end
```

OK, but the value should be a collection, furthermore, it should be indexed by the expected key:

```
it "should have an element that behaves like a collection" do
  @equiv_classes[:one].should respond_to(:[], :size, :entries)
end
```

Again, not much of a change is required:

```
def equivalence_classes(*keys, &block)
  {:one => []}
end
```

Next we can write some examples describing what's in the class... one small bit at a time. First it's size:

```
it "should have 5 items in the equiv class" do
  @equiv_classes[:one].should have(5).elements
end
```

And the resulting change:

```
def equivalence_classes(*keys, &block)
  {:one => [0, 0, 0, 0, 0]}
end
```

Next we can say something about what those 5 elements should be:

```
it "should have the expected elements" do
  @equiv_classes[:one].should include(0, 1, 2, 3, 4)
end
```

Again, a simple change:

```
def equivalence_classes(*keys, &block)
  {:one => [0, 1, 2, 3, 4]}
end
```

That's it for the first pass. We have the skeleton in place, and a literal solution for a simple case. This is often referred to as *faking it*. As we write example of more complex situations we will force our implementation to evolve into something more generic. That strategy is usually called *triangulation*.

3 Something a Little More Complex

We have examples for when the result is a single class. Now let's push a bit further with examples for a situation involved two classes. Partitioning numbers into sets of odd & even will do. Here's the before block and the first example to go with it:

```
describe "Dividing 1..10 into odd & even sets" do

  before(:each) do
    @equiv_classes = (1..10).to_a.equivalence_classes(:odd, :even) do |datum, key|
      case
      when key == :even && (datum & 1 == 0): true
      when key == :odd && (datum & 1 == 1): true
      else false
      end
    end
  end

  it "should have two classes" do
    @equiv_classes.should have(2).elements
  end
end
```

Again, we can largely fake this:

```
def equivalence_classes(*keys, &block)
  classes = {}
  keys.each {|key| classes[key] = self.entries}
  classes
end
```

The next example will be a bit meatier, and take a bit bigger of a step that the earlier examples:

```
it "should have the right even values" do
  @equiv_classes[:even].should have(5).elements
  @equiv_classes[:even].should include(2, 4, 6, 8, 10)
end
```

To satisfy this, we'll have to have our code do some work:

```
def equivalence_classes(*keys, &block)
  classes = {}
  keys.each {|key| classes[key] = []}
  self.each do |element|
    keys.each do |key|
      classes[key] << element if yield(element, key)
    end
  end
  classes
end
```

The next logical step is to give an example of defining the other class:

```

it "should have the right odd values" do
  @equiv_classes[:odd].should have(5).elements
  @equiv_classes[:odd].should include(1, 3, 5, 7, 9)
end

```

This time everything stays green. This should cause us to think.. *Is out example correct?*. Yes, it is... this is simply the flip side of the previous example.

4 The Final Requirement

So now, it's time to move on to some examples of overlapping classes that I mentioned early on. The situation I came up with is to divide up the numbers 1..10 into sets that are evenly divisible by 3, 2, and 1. This will raise the issue. For example, all the numbers are divisible by 1, and 6 is also divisible by 2 and 3. For these examples, we want the higher divisor to take priority.

So our first example (and the context) will just do a sanity check on the fact that the right number of result collections are being generated:

```

describe "When classes are overlapping, partitioning " do

  before(:each) do
    @equiv_classes = (1..10).to_a.equivalence_classes(3, 2, 1) do |datum, divisor|
      datum.modulo(divisor).zero?
    end
  end

  it "should have a class for each id" do
    @equiv_classes.should have(3).elements
    @equiv_classes[3].should_not be_nil
    @equiv_classes[2].should_not be_nil
    @equiv_classes[1].should_not be_nil
  end
end

```

This, as expected, is fine. On to the next example:

```

it "should have 3, 6, and 9 in the '3' class" do
  @equiv_classes[3].should have(3).elements
  @equiv_classes[3].should include(3, 6, 9)
end

```

This is fine as well... no surprise. Next we'll force the code to deal with the fact that 6 should not be in the "2" result set, since it is already in the "2" set:

```

it "should have 2, 4, 8, and 10 in the '2' class (NOT 6 since it's in the '3' class)" do
  @equiv_classes[2].should have(4).elements
  @equiv_classes[2].should include(2, 4, 8, 10)
end

```

As anticipated, that caused a problem. The approach I took was to break out of the inner loop (looking for the bucket for the current element) when it's home was found:

```

def equivalence_classes(*keys, &block)
  classes = {}
  keys.each {|key| classes[key] = []}
  self.each do |element|
    keys.each do |key|
      next unless yield(element, key)
      classes[key] << element
      break
    end
  end
end

```

```

    end
  end
  classes
end

```

And we write one final example to cover off the final case:

```

it "should have 1, 5, and 7 in the '1' class (the rest are in either the '2' or '3' classes)" do
  @equiv_classes[1].should have(3).elements
  @equiv_classes[1].should include(1, 5, 7)
end

```

This works with the code as is.

5 What If Everything Doesn't Fit?

In the previous examples, everything in the source collection fit into at least one result class. What if that's not the case? There could be leftover elements that don't fit in any result class. Conversely, there could be empty result classes. Let's write some examples to nail that down. Note that since I don't anticipate changing the code, I'll do them all at once. If any fail, I'll back up and work more incrementally.

```

describe "When something doesn't fit into any class" do
  before(:each) do
    @equiv_classes = (1..10).to_a.equivalence_classes(3, 2) do |datum, divisor|
      datum.modulo(divisor).zero?
    end
  end

  it "should have a class for each id" do
    @equiv_classes.should have(2).elements
    @equiv_classes[3].should_not be_nil
    @equiv_classes[2].should_not be_nil
  end

  it "should have 3, 6, and 9 in the '3' class" do
    @equiv_classes[3].should have(3).elements
    @equiv_classes[3].should include(3, 6, 9)
  end

  it "should have 2, 4, 8, and 10 in the '2' class" do
    @equiv_classes[2].should have(4).elements
    @equiv_classes[2].should include(2, 4, 8, 10)
  end
end

describe "When there a class that nothing fits into" do
  before(:each) do
    @equiv_classes = (1..10).to_a.equivalence_classes(11, 2, 1) do |datum, divisor|
      datum.modulo(divisor).zero?
    end
  end

  it "should have a class for each id" do
    @equiv_classes.should have(3).elements
    @equiv_classes[11].should_not be_nil
  end
end

```

```

    @equiv_classes[2].should_not be_nil
    @equiv_classes[1].should_not be_nil
  end

  it "should have nothing in the '11' class" do
    @equiv_classes[11].should be_empty
  end

  it "should have 2, 4, 6, 8, and 10 in the '2' class" do
    @equiv_classes[2].should have(5).elements
    @equiv_classes[2].should include(2, 4, 6, 8, 10)
  end

  it "should have 1, 3, 5, 7, and 9 in the '1' class" do
    @equiv_classes[1].should have(5).elements
    @equiv_classes[1].should include(1, 3, 5, 7, 9)
  end
end
end

All good.

```

5.1 Are we there yet?

We could go on to write more examples, but this should suffice... at least for now. With experience you will develop a sense of how many examples are enough. There are tools such as Heckle and rCov that give you a more objective opinion of whether you have a less than adequate amount of examples.

If we discover a problem later we can write examples to illustrate the problem, and drive to a solution.

6 Cleanup

One of the great benefits of working this way (whether you call it TDD or BDD) is that we now have a suite of regression tests that will let us know when we change the behaviour of the code. This is *great* because it allows us to aggressively refactor without having to work through all permutations of our changes before we make them. If everything is green before & after your change, it's safe. Of course, this requires us to be running the examples frequently. This is where a tool like Autotest comes into play. It will sit in the background, running the examples whenever a spec or code file changes.

For example, I don't like the first two lines that initialize the set of result classes:

```

classes = {}
keys.each {|key| classes[key] = []}

```

We can collapse that by using inject to build the initialized Hash:

```

classes = keys.inject({}) {|map, key| map[key] = []; map}

```

Note that we need to append `; map` since `map[key] = []` will return the value, in this case an empty array.

Our solution is now:

```

def equivalence_classes(*keys, &block)
  classes = keys.inject({}) {|map, key| map[key] = []; map}
  self.each do |element|
    keys.each do |key|
      next unless yield(element, key)
      classes[key] << element
      break
    end
  end
  classes
end

```

Since we have a working solution and can tell when it breaks, we can look for a more idiomatic solution. The following was suggested by Christian Neukirchen, which I really like:

```
def equivalence_classes(*keys, &block)
  to_partition = self
  classes = {}
  keys.each do |key|
    fitting, to_partition = to_partition.partition { |e| yield(e, key) }
    classes[key] = fitting
  end
  classes
end
```

This conforms to all the examples and is a much shorter and more idiomatic Ruby solution. Can we do better? Sure, let's use `inject` to build up the result and convert it into a `Hash` to be returned:

```
def equivalence_classes(*keys, &block)
  to_partition = self
  Hash[*keys.inject([]) do |result, key|
    fitting, to_partition = to_partition.partition { |e| yield(e, key) }
    result + [key, fitting]
  end]]
end
```

That might be a bit too busy. Another solution is:

```
def equivalence_classes(*keys, &block)
  classes = {}
  keys.inject(self) do |items, key|
    classes[key], rest = items.partition { |e| yield(e, key) }
    rest
  end
  classes
end
```

I like this one, and will stop here.

There's always a risk of tuning a solution too much. It's a balancing act between cleverness and clarity. The balance point is different for different people at different times (as their familiarity with a language increases, for example). But with examples in place, we are free to explore the solution space.

7 Thanks

Thanks to the kind folks who had a look at the drafts of this article: David Chelimsky, Rick Bradley, and Christian Neukirchen.

Special thanks to Christian for instigating and contributing to the final refactorings.