6

FUNCTIONALISM WITH BLOCKS AND PROCS

Ruby has two main ancestors: Smalltalk and Lisp.[1] From Smalltalk, Ruby gets its heavy object orientation, which we've explored in some depth up to this point. From Lisp it derives several ideas from *functional programming*, which is a very mathematically inclined approach to programming with a few notable characteristics. First, variables tend to be defined once, without having their values changed later on. Additionally, functions tend to be simple, abstract, and used as building blocks for other functions; the line between *functions*, which perform operations, and *data*, on which functions operate, is often blurry, compared with non-functional approaches. Functions also tend to do their work by returning values, rather than having side effects—in Ruby terms, methods that end with an exclamation point are less common.

Ruby's support for functional programming is extensive and exciting. Let's dive in.

---

[1] This is a potentially contentious statement. At a RubyConf, I once asked Matz which other languages he thought were most influential on Ruby. His response was "Smalltalk and Common Lisp". Other folks in the Ruby community (many of them ex-Perl users) stress Ruby's clear similarity to Perl. Probably the safest statement is that Ruby descends from Smalltalk and Lisp, and while it's a lot like Perl, Perl is more like an aunt or uncle.

#20 Our First lambda (make_incrementer.rb)

This script explores how Ruby creates functions that should be treated as objects. Every "thing" in Ruby is an object, so the notion of treating functions as objects is not conceptually odd. In Ruby, we do this with the command `lambda`, which takes a block. Let's look at that in irb.

```
irb(main):001:0> double_me = lambda { |x| x * 2 }
=> #<Proc:0xb7d1f890@(irb):1>
irb(main):002:0> double_me.call(5)
=> 10
```

You can see by the return value of line one that the result of calling `lambda` is an instance of class Proc. *Proc* is short for *procedure*, and while most objects are defined by what they *are*, Procs can be thought of primarily as defined by what they *do*. Procs have a method called *call*, which tells that Proc instance to do whatever it does. In our irb example, we have a Proc instance called `double_me` that takes an argument and returns that argument, times two. On line two, we see that feeding the number *5* into `double_me.call` results in a return value of *10*, just as you would expect. It is easy to create other Procs that do other operations.

```
irb(main):003:0> triple_me = lambda { |x| x * 3 }
=> #<Proc:0xb7d105bc@(irb):3>
irb(main):004:0> triple_me.call(5)
=> 15
```

Since Procs are objects, just like everything else in Ruby, we can treat them like any other object. They can be the returned value of a method, either the key or value of a Hash, arguments to other methods, and whatever else any object can be. Let's look at the script that demonstrates this.

The Code

Procs

Calling Procs

```
#!/usr/bin/env ruby
# make_incrementer.rb

❶ def make_incrementer(delta)
    return lambda { |x| x + delta }
  end

❷ incrementer_proc_of = Hash.new()
  [10, 20].each do |delta|
    incrementer_proc_of[delta] = make_incrementer(delta)
  end

❸ incrementer_proc_of.each_pair do |delta,incrementer_proc|
    puts "#{delta} + 5 = #{incrementer_proc.call(5)}\n"
  end
```

```
❹ puts

❺ incrementer_proc_of.each_pair do |delta,incrementer_proc|
❻   (0..5).to_a.each do |other_addend|
      puts "#{delta} + #{other_addend} = " +
        incrementer_proc.call(other_addend) + "\n"
    end
  end
```

The each_pair Method

How It Works

At ❶ we define a method called make_incrementer. It takes a single argument called delta and returns a Proc (created via lambda) that adds delta to something else, represented by x. What is that something else? We don't know yet. That is precisely the point of this method—it allows us to define an operation that can be performed multiple times using different parameters, just like any other function.

We can see how this is useful in the rest of this script. At ❷ we define a new Hash called incrementer_proc_of. For each of the values 10 and 20, we make an incrementer (using either 10 or 20 for the value of delta in the make_incrementer method) and assign the resulting Proc into the incrementer_proc_of Hash. Starting at ❸, we read each delta and Proc pair out of the Hash using the each_pair method and then use puts to print a line describing that delta value and the result of calling its Proc with the argument of 5.

We ❹ print a spacer with puts (just for ease of reading the output), and finally ❺ output another set of data. This time we add another loop for a value called other_addend; this is a variable that serves a role analogous to our static value of 5 in the loop (❸). Let's run this program with ruby -w make_incrementer.rb and look at the output.

The Results

```
20 + 5 = 25
10 + 5 = 15

20 + 0 = 20
20 + 1 = 21
20 + 2 = 22
20 + 3 = 23
20 + 4 = 24
20 + 5 = 25
10 + 0 = 10
10 + 1 = 11
10 + 2 = 12
10 + 3 = 13
10 + 4 = 14
10 + 5 = 15
```

The first two lines before the empty line show the output of the first loop (with the static value of `5` for the addend), while the rest of the output shows the result of the second loop, which uses the `other_addend` variable. Notice also that `each_pair` does not order by key, which is why my output has the delta value of `20` appearing first. Depending on your implementation of Ruby, you might see a `delta` of `10` first.

Now you know how to create Procs. Let's learn how to use them for something more useful than just demonstrating themselves.

#21 Using Procs for Filtering (matching_members.rb)

So far, we've seen that to create a Proc, we call `lambda` with a block describing what that Proc should do. This would lead you to believe that there is a special relationship between Procs and blocks, which there is. Our next script demonstrates how to use Procs in place of blocks.

The Code

Procs as Arguments

```
#!/usr/bin/env ruby
# matching_members.rb

=begin rdoc
Extend the built-in <b>Array</b> class.
=end
class Array

=begin rdoc
Takes a <b>Proc</b> as an argument, and returns all members
matching the criteria defined by that <b>Proc</b>.
=end
❶   def matching_members(some_proc)
      find_all { |i| some_proc.call(i) }
    end

  end

❷ digits = (0..9).to_a
  lambdas = Hash.new()
  lambdas['five+']   = lambda { |i| i >= 5 }
  lambdas['is_even'] = lambda { |i| (i % 2).zero? }

❸ lambdas.keys.sort.each do |lambda_name|
❹   lambda_proc  = lambdas[lambda_name]
❺   lambda_value = digits.matching_members(lambda_proc).join(',')
❻   puts "#{lambda_name}\t[#{lambda_value}]\n"
  end
```

How It Works

In this script, we open the Array class in order to add a new method called `matching_members` (❶). It takes a Proc (creatively called `some_proc`—see the note below) as an argument and returns the result of calling `find_all`, which (as its name suggests) finds all members for which the block is true. In this case, the condition in the block is the result of calling the Proc argument on the Array with the Array member in question as the argument to `call`. After we finish defining our new method, we set up our `digits` Array and our Procs with appropriate names in the `lambdas` Hash at ❷.

NOTE *Some of my co-workers make fun of the variable and method names I use—like* `some_proc`*, for example. I think names should either be very specific, like* `save_rates_to_local_file!`*, or explicitly generic, like* `some_proc`*. For truly generic operations, I often use variable names like* `any_proc` *or* `any_hash`*, which tell you explicitly that the operations being performed on them are meant to be useful for any Proc or Hash.*

At ❸, we loop through each sorted `lambda_name`, and at ❹ we extract each Proc out as a variable called `lambda_proc`. We then `find_all` members of the `digits` Array that match the condition described by that Proc at ❺ and `puts` an appropriate message at ❻.

Running the Script

Let's see it in action with `ruby -w matching_members.rb`.

The Results

```
five+   [5,6,7,8,9]
is_even [0,2,4,6,8]
```

In each case, we filter the members of the `digits` Array based on some specific conditions. Hopefully, you'll find that the names I chose for each Proc match what that Proc does. The `five+` Proc returns `true` for any argument that is five or greater.[2] We see that the results of calling `five+` on each digit in turn returns the correct digits. Similarly, the `is_even` Proc filters its input, only returning `true` for arguments that are even, where *evenness* is defined as having a modulus two equal to zero. Again, we get the correct numbers.

What happens when we want to filter based on multiple criteria? We could filter once with one Proc, assign that result into an Array, and then filter that result by the second criterion. That's perfectly valid, but what if we have an unknown number of filtering conditions? We want a version of `matching_members` that can take an arbitrary number of Procs. That's our next script.

#22 Using Procs for Compounded Filtering (matching_compound_members.rb)

In this script, we'll filter Arrays using an arbitrary number of Procs. As before, we'll open up the Array class, this time adding two methods. Again, we'll filter digits based on simple mathematical tests. Let's take a look at the source code and see what's different.

---

[2] It does this by implicit Boolean evaluation of the expression `i >= 5`.

The Code

```
#!/usr/bin/env ruby
# matching_compound_members.rb

=begin rdoc
Extend the built-in <b>Array</b> class.
=end
class Array

=begin rdoc
Takes a block as an argument and returns a list of
members matching the criteria defined by that block.
=end
```

Block Arguments

```
❶   def matching_members(&some_block)
      find_all(&some_block)
    end

=begin rdoc
Takes an <b>Array</b> of <b>Proc</b>s as an argument
and returns all members matching the criteria defined
by each <b>Proc</b> via <b>Array.matching_members</b>.
Note that it uses the ampersand to convert from
<b>Proc</b> to block.
=end
❷   def matching_compound_members(procs_array)
      procs_array.map do |some_proc|
        # collect each proc operation
❸       matching_members(&some_proc)
❹     end.inject(self) do |memo,matches|
        # find all the intersections, starting with self
        # and whittling down until we only have members
        # that have matched every proc
```

Array Intersections

```
❺       memo & matches
      end
❻   end

end

# Now use these methods in some operations.
❼ digits = (0..9).to_a
lambdas = Hash.new()
lambdas['five+']   = lambda { |i| i if i >= 5 }
lambdas['is_even'] = lambda { |i| i if (i % 2).zero? }
lambdas['div_by3'] = lambda { |i| i if (i % 3).zero? }

lambdas.keys.sort.each do |lambda_name|
  lambda_proc   = lambdas[lambda_name]
  lambda_values = digits.matching_members(&lambda_proc).join(',')
❽ puts "#{lambda_name}\t[#{lambda_values}]\n"
end

❾ puts "ALL\t[#{digits.matching_compound_members(lambdas.values).join(',')}]"
```

How It Works

We start by defining a method called `matching_members` (❶), just as before. However, this time our argument is called `some_block` instead of `some_proc`, and it is preceded by an ampersand. Why?

Blocks, Procs, and the Ampersand

The ampersand (`&`) is Ruby's way of expressing blocks and Procs in terms of each other. It's very useful for arguments to methods, as you might imagine. *Blocks*, you may remember, are simply bits of code between delimiters such as braces (`{ "I'm a block!" }`) or the `do` and `end` keywords (`do "I'm also a block!" end`). *Procs* are objects made from blocks via the `lambda` method. Either of them can be passed into methods, and the ampersand is the way to use one as the other. Let's test this in irb.

& Notation for Blocks and Procs

```
irb(main):001:0> class Array
irb(main):002:1> def matches_block( &some_block )
irb(main):003:2> find_all( &some_block )
irb(main):004:2> end
irb(main):005:1> def matches_proc( some_proc )
irb(main):006:2> find_all( &some_proc )
irb(main):007:2> end
irb(main):008:1> end
=> nil
```

We open the Array class and add a method called `matches_block`; this method takes a block (with an ampersand prefix), effectively duplicating the behavior of the existing `find_all` method, which it calls. We also add another method called `matches_proc` that calls `find_all` again, but takes a Proc this time. Then we try them out.

```
irb(main):009:0> digits = (0..9).to_a
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
irb(main):010:0> digits.matches_block { |x| x > 5 }
=> [6, 7, 8, 9]
irb(main):011:0> digits.matches_proc( lambda { |x| x > 5 } )
=> [6, 7, 8, 9]
```

The `matches_block` method dutifully takes a block and passes it along to the `find_all` method, transforming it along the way with the ampersand—once on input and again when passed to `find_all`. The `matches_proc` method takes a Proc and passes that on to `find_all`, but it only needs to transform with the ampersand once.

You might think that we could omit the ampersand and just treat a block argument as a standard variable, like in irb below.

```
irb(main):001:0> class Array
irb(main):002:1> def matches_block( some_block )
irb(main):003:2> find_all( some_block )
irb(main):004:2> end
```

```
irb(main):005:1> end
=> nil
irb(main):006:0> digits = (0..9).to_a
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
irb(main):007:0> digits.matches_block { |x| x > 5 }
ArgumentError: wrong number of arguments (0 for 1)
        from (irb):7:in `matches_block'
        from (irb):7
        from :0
```

That doesn't work, as you see. Ruby keeps track of the number of arguments that a given method, block, or Proc expects (a concept called *arity*) and complains when there is a mismatch. Our irb example expected a "real" argument, not just a block, and complained when it didn't get one.

NOTE *The gist of the ArgumentError is that blocks are akin to "partial" or "unborn" blocks and need the* `lambda` *method to be made into full-fledged Procs, which can be used as real arguments to methods. Some methods, like* `find_all`*, can handle block arguments, but these block arguments are treated differently than regular arguments and don't count toward the number of "real" arguments. We'll cover this later when we discuss the* `willow_and_anya.rb` *script. For now, note that our new version of* `matching_members` *takes a block instead of a Proc.*

Filtering with Each Proc via map

We also define a new method called `matching_compound_members` at ❷. The `matching_compound_members` method takes an Array argument called `procs_array` and maps a call to `matching_members` onto each of `procs_array`'s Proc elements; this transforms the elements into blocks with the ampersand at ❸ while doing the mapping. This results in an Array, each of whose members is an Array containing all members of the original Array that match the conditions defined by the Proc. Confused? Take a look in irb.

```
irb(main):001:1> class Array
irb(main):002:1> def matching_compound_members( procs_array )
irb(main):003:2> procs_array.map do |some_proc|
irb(main):004:3* find_all( &some_proc )
irb(main):005:3> end
irb(main):006:2> end
irb(main):007:1> end
=> nil
irb(main):008:0> digits.matching_compound_members( [ lambda { |x| x > 5 },
lambda { |x| (x % 2).zero? }])
=> [[6, 7, 8, 9], [0, 2, 4, 6, 8]]
```

On lines one through seven, we add a shortened version of `matching_members` to all Arrays. We call it on line eight, and find that the result is an Array of Arrays. The first sub-array is all digits greater than five—the result of the first Proc. The second sub-array is all even digits—the result of the second Proc. That's what we have at the end of the `map` (❹) inside `matching_compound_members`.

Finding the Intersections with inject

We don't stop there. Next we call our old friend the `inject` method on that Array of Arrays. You may remember that `inject` performs an operation successively and has a memory for intermediate results. That will be very useful for us. The `inject` method takes an optional non-block element for the initial state of its memory. In our script we use `self` (❹), meaning that the memory state will be the `self` Array as it exists prior to any filtering. We also say that each member of the Array resulting from the `map` operation will be called `matches`. This makes sense because the `matches` variable represents members of the initial Array that were found to match the Proc used for that particular stage of the `map` operation.

Array Intersections

At ❺, we call a method we haven't seen before on `memo`. This method happens to be expressed with the ampersand character, but it has nothing to do with converting blocks and Procs into each other; it has more to do with set math.

```
irb(main):001:0> digits = (0..9).to_a
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
irb(main):002:0> evens = digits.find_all { |x| (x % 2).zero? }
=> [0, 2, 4, 6, 8]
irb(main):003:0> digits & evens
=> [0, 2, 4, 6, 8]
irb(main):004:0> half_digits = digits.find_all { |x| x < 5 }
=> [0, 1, 2, 3, 4]
irb(main):005:0> evens & half_digits
=> [0, 2, 4]
```

Can you guess what this ampersand means? It represents the intersection of two composite data sets. It basically means *Find all members of myself that also belong to this other thing*. When we call it within our `inject`, we ensure that once a given Array element fails one test, it no longer appears as a candidate for the next test. This happens because the memory of the `inject` method (represented by the variable called `memo`) is automatically set to the return value of each iteration of the `inject` method. At ❻, when we're done with all of our `mapping` and `injecting`, we're left with only those members of the original Array that pass the tests defined by every single Proc in the `procs_array` argument. Since Ruby returns the last expression evaluated in a method, `matching_compound_members` returns an Array of all members of `self` that pass every test represented by the members of `procs_array`.

After some setup at ❼ similar to that for the previous script, we output results using `puts` at both ❽ and ❾. Let's see it in action.

The Results

```
div_by3 [0,3,6,9]
five+   [5,6,7,8,9]
is_even [0,2,4,6,8]
ALL     [6]
```

We call each of these filtering Procs on the digits from zero to nine, getting the correct members each time. We finally output the prefix `ALL` followed by the members that pass all the tests. The number six is the only digit from zero to nine that is divisible by three, is greater than or equal to five, and is even. Therefore, it is the only member of the final output.

Hacking the Script

Try defining your own Procs using `lambda`. You can add them to the section at ❼ or replace some of the existing Procs. Feel free to alter the range used to create the `digits` Array as well. A larger range of values in `digits` could help demonstrate more complex relationships among a greater number of filtering Procs.

#23 Returning Procs as Values (return_proc.rb)

Let's look at a further demonstration of how to use Procs as data generated by another function. It's very similar to the `make_incrementer.rb` script.

The Code

Procs as Hash Values

Making Strings with %Q

```
#!/usr/bin/env ruby
# return_proc.rb

❶ def return_proc(criterion, further_criterion=1)

    proc_of_criterion = {
      'div_by?' => lambda { |i| i if (i % further_criterion).zero? },
      'is?'     => lambda { |i| i == further_criterion }
    }

    # allow 'is_even' as an alias for divisible by 2
❷   return return_proc('div_by?', 2) if criterion == ('is_even')

❸   proc_to_return = proc_of_criterion[criterion]
    fail "I don't understand the criterion #{criterion}" unless proc_to_return
    return proc_to_return

  end

❹ require 'boolean_golf.rb'

  # Demonstrate calling the proc directly
❺ even_proc = return_proc('is_even') # could have been ('div_by', 2)
  div3_proc = return_proc('div_by?', 3)
  is10_proc = return_proc('is?', 10)
❻ [4, 5, 6].each do |num|
    puts %Q[Is #{num} even?: #{even_proc[num].true?}]
    puts %Q[Is #{num} divisible by 3?: #{div3_proc[num].true?}]
```

The `inspect` **Method**

```
  puts %Q[Is #{num} 10?: #{is10_proc[num].true?}]
❼ printf("%d is %s.\n\n", num, even_proc[num].true? ? 'even' : 'not even')
end

# Demonstrate using the proc as a block for a method
❽ digits = (0..9).to_a
even_results = digits.find_all(&(return_proc('is_even')))
div3_results = digits.find_all(&(return_proc('div_by?', 3)))
❾ puts %Q[The even digits are #{even_results.inspect}.]
puts %Q[The digits divisible by 3 are #{div3_results.inspect}.]
puts
```

The Results

If we call this with the command `ruby -w return_proc.rb`, we get the following output, all of which is true.

```
Is 4 even?: true
Is 4 divisible by 3?: false
Is 4 10?: false
4 is even.

Is 5 even?: false
Is 5 divisible by 3?: false
Is 5 10?: false
5 is not even.

Is 6 even?: true
Is 6 divisible by 3?: true
Is 6 10?: false
6 is even.

The even digits are [0, 2, 4, 6, 8].
The digits divisible by 3 are [0, 3, 6, 9].
```

How It Works

We define a method called `return_proc` starting at ❶ that takes a mandatory `criterion` and an optional `further_criterion`, assumed to be one. It then defines a Hash called `proc_of_criterion` with keys that match a specific criterion and values that are Procs corresponding to each criterion. It then allows a caller to use an alias `is_even` to mean *Divisible by two* at ❷. It does this by recursively calling itself with the arguments `div_by?` and `2` when the alias is used.

Assuming that the `is_even` alias is not used, the method tries to read the appropriate Proc to use at ❸; it fails if it gets a criterion it doesn't understand.[3] If it gets past this point, we know that the method understands its criteria, because it found a Proc to use. It then returns that Proc, appropriately called `proc_to_return`.

[3] Were you to modify or extend this method, you could simply add more options to the `proc_of_criterion` Hash.

We now know that `return_proc` lives up to its name and returns a Proc. Let's use it. At ❹, we `require` one of our first scripts, `boolean_golf.rb`. You may recall that that script adds the methods `true?` and `false?` to every object. This will come in handy for our next few lines. At ❺, we define three Procs that can test numbers for certain conditions. We then use those Procs within the `each` block starting at ❻. For each of the Integers `4`, `5`, and `6`, we test for evenness, being divisible by three, and being equal to ten. We also use both the `printf` command that we saw in the `line_num.rb` script and the main ternary operator, both of which happen at ❼.

Proc.call(args) vs. Proc[args]

Notice that we call our Procs with a different syntax here—we don't use the `call` method at all. We can simply put whatever arguments we would use inside square brackets, and it's just like using the `call` method. Let's verify this in irb.

```
irb(main):001:0> is_ten = lambda { |x| x == 10 }
=> #<Proc:0xb7d0c8a4@(irb):1>
irb(main):002:0> is_ten.call(10)
=> true
irb(main):003:0> is_ten[10]
=> true
irb(main):004:0> is_ten.call(9)
=> false
irb(main):005:0> is_ten[9]
=> false
```

I chose to use the bracket syntax in these examples for the sake of brevity. So far, I've shown how to use Procs that have been returned directly from the `return_proc` method. But we can also do other things, such as converting between blocks and Procs.

Using Procs as Blocks

From ❽ to the end of the script, we see how we can cast the output of `return_proc` (which we know to be a Proc) into a block with the ampersand without ever storing the Proc in a variable. After defining our usual `digits` Array, we call `find_all` twice, assigning the results into `even_results` and `div3_results`, respectively. Remember that `find_all` takes a block. The ampersand can convert any expression that evaluates to a Proc into a block, and `(return_proc('is_even'))` is an expression that returns (evaluates to) a Proc. Therefore, we can coerce (or cast) the expression `(return_proc('is_even'))` into a perfectly valid block for `find_all`. We do this, outputting the results via `puts` at ❾.

The inspect Method

Notice that we call a new method called `inspect` on each set of results to retain the brackets and commas that we normally associate with members of Arrays. The `inspect` method returns a String representation of whatever object it's called on. It is slightly different from the to_s method we've already seen. Let's check that out in irb.

```
irb(main):001:0> digits = (0..9).to_a
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
irb(main):002:0> digits.to_s
=> "0123456789"
irb(main):003:0> digits.inspect
=> "[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]"
```

You can see that the output of inspect is a bit prettier than the output of to_s. It also retains more information about what type of object it was called on.

You should now be pretty comfortable with calling Procs, passing them around, reading them out of Hashes, and converting them to and from blocks, whether with a lambda or when passing around to methods. Now let's look at nesting lambdas within other lambdas.

#24 Nesting lambdas

Let's review Procs for a bit. Procs are just functions that can be treated as data, what functional programming languages call *first-class functions*. Functions can create Procs; we saw that both make_incrementer and return_proc return Procs of different sorts. Given all that, what prevents us from making a Proc that returns another Proc when called? Nothing at all.

In the make_exp example below, we create specific versions of Procs that raise an argument to some specified power. That power is the exp argument taken by the outer lambda, which is described as a *free variable* because it is not an explicit argument to the inner lambda.

The inner lambda, which is returned, has a *bound variable* called x. It is bound because it is an explicit argument to that inner lambda. That variable x is the number that will be raised to the specified power. This example is short, and the returned value at each stage is very important, so we'll do this entirely in irb.

The Code

Nested Lambdas

```
irb(main):001:0> digits = (0..9).to_a
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
irb(main):002:0> make_exp_proc = lambda { |exp| lambda { |x| x ** exp } }
=> #<Proc:0xb7c97adc@(irb):2>
irb(main):003:0> square_proc = make_exp_proc.call(2)
=> #<Proc:0xb7c97b18@(irb):2>
irb(main):004:0> square_proc.call(5)
=> 25
irb(main):005:0> squares = digits.map { |x| square_proc[x] }
=> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

How It Works

We see up to this point that `make_exp_proc` is a Proc, which returns a Proc when called. That resulting Proc raises its argument to the exponent used in the initial call of `make_exp_proc`. Since in our example, we called `make_exp_proc` with `2`, we created a Proc that squares its argument, appropriately calling it `square_proc`. We also see that the squaring Proc can be used in a mapping operation onto the digits Array, and that it returns the correct squared values.

```
irb(main):006:0> cube_proc = make_exp_proc.call(3)
=> #<Proc:0xb7c97b18@(irb):2>
irb(main):007:0> cube_proc.call(3)
=> 27
irb(main):008:0> cubes = digits.map { |x| cube_proc[x] }
=> [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

We also see in the rest of the example that `make_exp_proc` is flexible and can take arguments other than `2`. It works perfectly well with an argument of `3`, producing a cubing Proc, which we can use in the same ways as the squaring Proc.

Up to this point, our Procs have tended to implement simple mathematical operations, like addition, multiplication, or exponentiation. But Procs are functions like any other, and they can output any type of value. Let's move on to the next script, which uses Procs that manipulate Strings.

#25 Procs for Text (willow_and_anya.rb)

As I was planning the functional programming chapter of this book, I was watching DVDs of Joss Whedon's *Buffy the Vampire Slayer*. I mention this because I had Procs and blocks on my brain, and I happened to encounter two very good candidates for text-based examples of `lambda` operations. In an episode called "Him," there is discussion of a "love spell", an "anti-(love spell) spell", and an "anti-(anti-(love spell) spell) spell". That's a great example of successive modifications via a simple function. In another episode called "Same Time, Same Place," there is a conversation that demonstrates simple variable substitution. Both are great examples of simple functions and are good venues to explore how Procs in Ruby differ based on how we choose to create them. Here's the source code.

NOTE *You obviously don't need to like Buffy to benefit from reading about these examples. The specific content that the scripts modify is essentially arbitrary.*

The Code

This code consists of three distinct files: one each for the two necessary classes, and one separate script meant to be directly executed.

The Him Class

```
#!/usr/bin/env ruby -w
# him.rb

❶ class Him

    EPISODE_NAME = 'Him'
    BASE         = 'love spell'

    ANTIDOTE_FOR = lambda { |input| "anti-(#{input}) spell" }

❷   def Him.describe()
      return <<DONE_WITH_HEREDOC

In #{EPISODE_NAME},
  Willow refers to an "#{ANTIDOTE_FOR[BASE]}".
  Anya mentions an "#{ANTIDOTE_FOR[ANTIDOTE_FOR[BASE]]}".
  Xander mentioning an "#{ANTIDOTE_FOR[ANTIDOTE_FOR[ANTIDOTE_FOR[BASE]]]}"
might have been too much.

DONE_WITH_HEREDOC
  end

end
```

Constant Procs

Class Methods

The SameTimeSamePlace Class

```
#!/usr/bin/env ruby -w
# same_time_same_place.rb

❸ class SameTimeSamePlace

    EPISODE_NAME = 'Same Time, Same Place'

=begin rdoc
This Hash holds various procedure objects. One is formed by the generally
preferred Kernel.lambda method. Others are created with the older Proc.new
method, which has the benefit of allowing more flexibility in its argument
stack.
=end
❹   QUESTIONS = {

      :ternary => Proc.new do |args|
        state    = args ? args[0] : 'what'
        location = args ? args[1] : 'what'
        "Spike's #{state} in the #{location}ment?"
      end,

      :unless0th => Proc.new do |*args|
        args = %w/what what/ unless args[0]
```

Flexible Arity with `Proc.new`

```
      "Spike's #{args[0]} in the #{args[1]}ment?"
    end,

    :nitems => Proc.new do |*args|
      args.nitems >= 2 || args.replace(['what', 'what'])
      "Spike's #{args[0]} in the #{args[1]}ment?"
    end,

    :second_or => Proc.new do |*args|
      args[0] || args.replace(['what', 'what'])
      "Spike's #{args[0]} in the #{args[1]}ment?"
    end,

    :needs_data => lambda do |args|
      "Spike's #{args[0]} in the #{args[1]}ment?"
    end

  }

❺ DATA_FROM_ANYA = ['insane', 'base']

❻ def SameTimeSamePlace.describe()

    same_as_procs = [
      SameTimeSamePlace.yield_block(&QUESTIONS[:nitems]),
      QUESTIONS[:second_or].call(),
      QUESTIONS[:unless0th].call(),
      SameTimeSamePlace.willow_ask,
    ]

        return <<DONE
In #{EPISODE_NAME},
  Willow asks "#{QUESTIONS[:ternary].call(nil)}",
  #{same_as_procs.map do |proc_output|
    'which is the same as "' + proc_output + '"'
    end.join("\n  ")
  }
  Anya provides "#{DATA_FROM_ANYA.join(', ')}", which forms the full question
  "#{SameTimeSamePlace.yield_block(DATA_FROM_ANYA, &QUESTIONS[:needs_data])}".

DONE
  end

=begin rdoc
Wrapping a lambda call within a function can provide
default values for arguments.
=end
❼ def SameTimeSamePlace.willow_ask(args = ['what', 'what'])
    QUESTIONS[:needs_data][args]
  end
```

```
=begin rdoc
Passing a block as an argument to a method
=end
❽  def SameTimeSamePlace.yield_block(*args, &block)
    # yield with any necessary args is the same as calling block.call(*args)
    yield(*args)
  end

end
```

The `yield` Method

The willow_and_anya.rb Script

```
#!/usr/bin/env ruby -w
# willow_and_anya.rb

%w[him same_time_same_place].each do |lib_file|
  require "#{lib_file}"
end

[Him, SameTimeSamePlace].each do |episode|
❾  puts episode.describe()
end
```

Arrays with `%w`

How It Works

This script performs some complex operations. Let's consider each class individually and then look at the separate script that uses them.

The Him Class: Creating Procs with lambda

We define a class called `Him` at ❶. It has three constants: its own `EPISODE_NAME`, a `BASE` item, and a `lambda` operation to create an `ANTIDOTE_FOR` something.[4] It has one class method called `Him.describe` (❷) that returns a long String constructed via a `here doc`. Remember that you can call a Proc with either `some_proc.call(args)` or `some_proc[args]`. In this case, we'll use the shorter bracket version again. We'll report that the character named Willow refers to the antidote for the base spell. Her associate Anya then mentions the antidote for that antidote. Whedon avoided yet another call to the antidote-creating Proc in his show, but our method will continue, outputting the antidote for the antidote for the antidote.

The SameTimeSamePlace Class: Alternatives to lambda for Creating Procs

Our next class explores more options. `SameTimeSamePlace` starts at ❸ and it defines a Hash constant called `QUESTIONS` right away at ❹. Its keys are Symbols, and its values are Procs. Up until now, we've always created Procs with the `lambda` method, but we know that Procs are instances of the class Proc. Traditionally, you can create an instance by calling the new method on a class. Let's try that in irb.

[4] I mentioned earlier in the book that `lambdas` can make excellent Class Constants. Now you can see that in action.

```
irb(main):001:0> is_even_proc1 = lambda { |x| (x % 2).zero? }
=> #<Proc:0xb7cb687c@(irb):1>
irb(main):002:0> is_even_proc2 = Proc.new { |x| (x % 2).zero? }
=> #<Proc:0xb7cacb4c@(irb):2>
irb(main):003:0> is_even_proc1.call(7)
=> false
irb(main):004:0> is_even_proc2.call(7)
=> false
irb(main):005:0> is_even_proc1.call(8)
=> true
irb(main):006:0> is_even_proc2.call(8)
=> true
```

That seems to work fine, and each Proc behaves as expected. In actual practice, there is little difference between Procs created via `lambda` and Procs created via `Proc.new`. `Proc.new` is a bit more flexible about how it handles arguments, which we'll soon see. For now, note that the value for the key `:ternary` in our `QUESTIONS` Hash at ❹ is a Proc that asks if someone named Spike has a certain `state` (which is neither already known nor static) in a certain `location` (which is also neither already known nor static).

NOTE *Don't be fooled by this script's surface-level silliness. It actually clarifies some very interesting behavior in Ruby's Procs with regard to arguments and arity. Later scripts that use these techniques for tasks that are more useful in the real world include scripts that convert temperatures and play audio files for a radio station.*

Flexible Arity for Proc.new

Next, we'll start exploring `Proc.new` more for the `:unless0th` Symbol key. You'll notice that the `*args` argument to this Proc has a preceding asterisk. This option is available to Procs created with `Proc.new`, but not to Procs created with `lambda`. It indicates that the argument with the asterisk is optional. Immediately inside the `:unless0th` Proc, we set the value of `args` if it has no value at the zeroth index; then we output the same question as the `:ternary` version. The only difference is that the `args` Array is optional for this version. Note also that we create our double `"what"` default Array with a `%w` with slash delimiters. This is a very handy way to create single-word Arrays.

For the `:nitems` Symbol key, we use an optional `*args` with `Proc.new` again. The only difference between this version and the `:unless0th` version is the way this tests `args`. In this version, we call the `nitems` method on the `args` Array, which returns the number of non-`nil` items. That number needs to be two or greater; if it isn't, that means we don't have enough elements, and so we will replace `args` with our default set of two `"what"`s, just as in the previous Procs.

For the `:second_or` Symbol key, we see yet another Proc within optional `args` created with `Proc.new`. This version simply tests whether or not the second item in the `args` Array can be read. If it cannot be read, we replace `args` just as in the `:nitems` version.

Finally, we create a Proc the way we always have, using `lambda`. Since arguments to `lambda` Procs are not optional, we identify this one with the Symbol `:needs_data`. Note that this makes the internals of the Proc simpler. It returns its output value, and we assume that it gets what it needs. After defining our Procs, the last of which needs data, we should probably have some data. Our source is Anya again, and we define her DATA_FROM_ANYA Array at ❺.

On to the method SameTimeSamePlace.describe at ❻. It takes no arguments and defines a local Array variable called same_as_procs. Its first element is the return value of calling SameTimeSamePlace.yield_block (defined at ❽) with an argument that is the Proc associated with the :nitems key in the QUESTIONS Hash. All of this is cast into a block with the ampersand. We haven't seen the yield_block method yet, but it takes two arguments: *args and &block. The first of these indicates *All of your regular arguments*, and the second means *Whatever block you got.*

Blocks, Arguments, and yield

Remember how I mentioned that blocks are not considered "real" arguments? Using an ampersand is the way to explicitly refer to the block used to call a method. Since we have the group of arguments, whatever they may be, and we have the block, we could call it via block.call(*args). That approach would work, but we have yet another alternative. Ruby has a method called yield that means *Call whichever block you received with whichever arguments are passed to yield*. When you get comfortable with this script, try replacing the yield line in yield_block with block.call(*args). It will not change the script's behavior at all. Let's verify some of this in irb.

```
irb(main):001:0> def yield_block(*args, &block)
irb(main):002:1> yield(*args)
irb(main):003:1> end
=> nil
irb(main):004:0> yield_block(0) { |x| x + 1 }
=> 1
irb(main):005:0> yield_block("I am a String") { |x| x.class }
=> String
irb(main):006:0> yield_block("How many words?") { |x| x.split(' ').nitems }
=> 3
irb(main):007:0> yield_block(0, 1) { |x,y| x == y }
=> false
irb(main):008:0> yield_block(0, 1) { |x,y| x < y }
=> true
```

Handy, isn't it? The yield_block method is completely generic, taking any number of regular arguments and any block and executing (or yielding) that block with those arguments. It's a very powerful technique.

Now we understand how our script is using the yield_block method within SameTimeSamePlace.describe (❻). The next two elements of same_as_procs are the return values of Procs pulled out of the QUESTIONS Hash with the call method. Our last element is the return value of SameTimeSamePlace.willow_ask (❼). This method provides a workaround for Procs created with lambda that need a specific number of arguments. willow_ask wraps a call to such a Proc within a traditional method that takes an optional argument. That argument is forcibly set to whatever the Proc expects before it ever gets to the Proc. This is another alternative for dealing with the arguments to a Proc.

That's it for the elements of our `same_as_procs` Array. Now let's use it. We `return` a long `here doc` String inside `SameTimeSamePlace.describe` (❻). This `here doc` String consists of several lines. The first calls the `QUESTIONS[:ternary]` Proc with one explicitly `nil` argument. This will cause our `state` and `location` variables to be set to their default values within the Proc. The next four lines of output are the result of mapping a String outputter onto the elements of `same_as_procs`. Remember that those elements are the `return` values of their respective Procs, not the Procs themselves. They have already been evaluated before being put into the Array.

The last few lines of the `here doc` report the data provided by Anya, which is defined as the constant Array `DATA_FROM_ANYA` (❺). We call the `yield_block` method, passing in `DATA_FROM_ANYA` as the "real" arguments and the value returned from `QUESTIONS[:needs_data]`, cast from a Proc into a block. Then we close our `here doc` and end the `SameTimeSamePlace.describe` method.

Using Both Him and SameTimeSamePlace in willow_and_anya.rb

The first thing we do in the main running script, `willow_and_anya.rb`, is `require` each `lib_file` needed. Then we cycle through each class, referred to by the name `episode`, and describe that episode (❾), implemented in each specific case, as already discussed.

Running the Script

Let's look at the output returned by executing `ruby -w willow_and_anya.rb`.

The Results

```
In Him,
  Willow refers to an "anti-(love spell) spell".
  Anya mentions an "anti-(anti-(love spell) spell) spell".
  Xander mentioning an "anti-(anti-(anti-(love spell) spell) spell) spell"
  might have been too much.

In Same Time, Same Place,
  Willow asks "Spike's what in the whatment?",
  which is the same as "Spike's what in the whatment?"
  which is the same as "Spike's what in the whatment?"
  which is the same as "Spike's what in the whatment?"
  which is the same as "Spike's what in the whatment?"
  Anya provides "insane, base", which forms the full question
  "Spike's insane in the basement?".
```

That's a lot of data about some pretty esoteric programming topics. Congratulations for sticking with me this far. If you're genuinely curious about how this all works, I have some questions for you to ponder.

Hacking the Script

How would you duplicate just the successive `lambda` outputs of `Him.describe` using `inject`? Here's what I came up with. Maybe you can find a better alternative.

```
def Him.describe2(iterations=3)
  (1..iterations).to_a.inject(BASE) do |memo,output|
    ANTIDOTE_FOR[memo]
  end
end
```

Another question you may find interesting is why the `describe` methods are attached to classes, rather than instances. The reason is that the `episode` variable at ❾ represents a class, not an instance. If we wanted to use instance methods, we would need to create an instance of either `Him` or `SameTimeSamePlace`, rather than just calling the `describe` method on each class directly.

Chapter Recap

What was new in this chapter?

- Creating Procs with `lambda`
- Using Procs as arguments to methods
- Using blocks as arguments to methods, including your own new methods
- Using Procs as first-class functions
- The `inspect` method
- Nesting `lambda`s within other `lambda`s
- `Proc.new`
- The `yield` method

I have a confession to make. I love object orientation for many programming tasks, but this chapter about Ruby's functional heritage was the most fun to write so far. Functional programming has been respected in academia for decades, and it is starting to get some well-deserved attention from folks in the computer programming industry and others who are just curious about what it can do. Now that we know some functional programming techniques, let's put them to use and even try to optimize them, which is the subject of our next chapter.