First topic: Dynamic and Static scoping. Consider the following:

```plaintext
int x = 0;
int f() { return x; }
int g() { int x = 1; return f(); }
```

Dynamic scoping: g() returns 1. Static scoping: g() returns 0.

```
(define y 3)
(define (f x) (+ x y))
(let ((x 10)
      (y 20))
  (f 100))
```

Dynamic scoping: f returns 120. Static scoping: f returns 103.

Static scoping: look at environment of definition
Dynamic scoping: look at environment of execution

Review of coerce: if 5+a is called, and 5 doesn’t know how to add an argument of type a, 5 does the following call:
a.coerce(self) #equivalent here to a.coerce(5)

So, in the a object, need a coerce method, which returns a new pair of things to try adding to get the result. The first return works because a+a already works. The second return works because a+5 already works.

```plaintext
def coerce(other)
    return [Section.new(other), self]
#could also use return [self, other]
end
```

Review: eval and bindings. Consider the following class:

```plaintext
class Foo
    def initialize
        @field = 43
    end
    def create_block
        Proc.new {}
    end
end
```
proc = Foo.new.create_block

#proc has with it the environment where the proc #was created, which is the environment inside the Foo class at the point of creation

puts eval("self.class", proc.binding)  #result is Foo. “self”, in the provided environment, refers to the Foo class

puts eval("@field", proc.binding)      #result is 43. “@field” in the provided environment is the field of Foo at the moment the proc was created.

-Grab bag of Ruby stuff

multiple arguments to a method:

    def foo(a, b, *args)
        args.each { .. }
    end

parallel assignment: i, j = j, i  #(swap with no temp)

Multiple return values:

    a, b = foo

    def foo
        return 5, 6
    end