

Functional techniques in Ruby

@erockenjew

The Big Idea

Functions are data too.

Strings are data

Numbers are data

Why can't functions be data?

Data can be manipulated. It can be composed of other data. Data can also be passed as parameters into functions, and returned from functions.

```
list = [1,2,3,4,5,6,7,8,9,10]
```

First lets step back and look at a common occurrence in all our programs, iteration.
list.each looks significantly different than the previous two examples
Point out for and in being language reserved words
while list.each is just a method invocation on list

```
list = [1,2,3,4,5,6,7,8,9,10]
```

in a language like Java, C, C++ you could write something like:

```
for(int i = 0; i < list.size(); i++){  
    puts list[i];  
}
```

First lets step back and look at a common occurrence in all our programs, iteration.
list.each looks significantly different than the previous two examples
Point out for and in being language reserved words
while list.each is just a method invocation on list

```
list = [1,2,3,4,5,6,7,8,9,10]
```

in a language like Java, C, C++ you could write something like:

```
for(int i = 0; i < list.size(); i++){  
    puts list[i];  
}
```

```
for i in list  
    puts i  
end
```

First lets step back and look at a common occurrence in all our programs, iteration. `list.each` looks significantly different than the previous two examples. Point out `for` and `in` being language reserved words while `list.each` is just a method invocation on `list`

```
list = [1,2,3,4,5,6,7,8,9,10]
```

in a language like Java, C, C++ you could write something like:

```
for(int i = 0; i < list.size(); i++){  
    puts list[i];  
}
```

```
for i in list  
    puts i  
end
```

```
list.each { |i| puts i }
```

First lets step back and look at a common occurrence in all our programs, iteration.
list.each looks significantly different than the previous two examples
Point out for and in being language reserved words
while list.each is just a method invocation on list

{ | i | puts i }

{ | i | puts i }

{ } # define the start and end of the block

{ | i | puts i }

{ } # define the start and end of the block

| i | # are the parameters of the block

{ | i | puts i }

{ } # define the start and end of the block

| i | # are the parameters of the block

puts i # is the body of the block

`{ | i | puts i }`

`}` *# define the start and end of the block*

`| i |` *# are the parameters of the block*

`puts i` *# is the body of the block*

```
def anon_function(i)
  puts i
end
```

What's in a name?

Block

Proc

closure

lambda

These anonymous functions can be called many different things. In Ruby they are mainly called blocks. In other languages and sometimes in Ruby people will call them closures. lambda means the same thing, comes from LISP.

```
# pseudo ruby code  
class Array  
  def each  
    for(i=0; i < self.size; i++)  
      yield self[i]  
    end  
  end  
end
```

yield is a ruby keyword that calls the block, passing in its arguments to the call of the code block

```
# pseudo ruby code  
class Array  
  def each  
    for(i=0; i < self.size; i++)  
      yield self[i]  
    end  
  end  
end
```

yield self[i]

yield is a ruby keyword that calls the block, passing in its arguments to the call of the code block

```
list.each {|i| puts i }
```

```
yield 1 => { |1| puts 1 }  
yield 2 => { |2| puts 2 }  
yield 3 => { |3| puts 3 }  
yield 4 => { |4| puts 4 }  
yield 5 => { |5| puts 5 }  
yield 6 => { |6| puts 6 }  
yield 7 => { |7| puts 7 }  
yield 8 => { |8| puts 8 }  
yield 9 => { |9| puts 9 }  
yield 10 => { |10| puts 10 }
```

```
class Array
  # still pseudocode
  def each(block)
    for(i=0; i < self.size; i++)
      block.call(self[i])
    end
  end
end
```

```
block.call(self[i])
```

```
block.call(1) => { |1| puts 1 }
block.call(2) => { |2| puts 2 }
block.call(3) => { |3| puts 3 }
block.call(4) => { |4| puts 4 }
block.call(5) => { |5| puts 5 }
block.call(6) => { |6| puts 6 }
block.call(7) => { |7| puts 7 }
block.call(8) => { |8| puts 8 }
block.call(9) => { |9| puts 9 }
block.call(10) => { |10| puts 10 }
```

.call has to go somewhere

```
bLock.class #=> Proc
```

```
list.each { |i| puts i }
```

==

```
b = Proc.new { |i| puts i }  
list.each(&b)
```

```
b = Proc.new { |i| puts i }  
list.each(&b)
```

==

```
b = lambda { |i| puts i }  
list.each(&b)
```

==

```
b = proc { |i| puts i }  
list.each(&b)
```

lambda and proc are just shortcuts for Proc.new, although with a few minor differences.

Quick Review

- Functions are “First Class” objects
- Higher Order Procedures accept functions as arguments

Now for the why?

- Because First Class functions allow us to abstract and combine patterns of computation

pattern of computation

where's the iteration code?

No more iteration. With each, we've abstracted out the pattern of iteration. Now your code can focus on what makes it special, or, how it changes, which is the body of the loop.

No more for loop controlled by a counter variable

pattern of computation

```
list.each { |i| puts i }
```

where's the iteration code?

No more iteration. With each, we've abstracted out the pattern of iteration. Now your code can focus on what makes it special, or, how it changes, which is the body of the loop.

No more for loop controlled by a counter variable

another example

```
def even?(num)
  num % 2 == 0
end
```

```
def reject_evens(list)
  return_list = Array.new
  list.each do |item|
    return_list << item unless even?(item)
  end
  return_list
end
```

```
def reject_odds(list)
  return_list = Array.new
  list.each do |item|
    return_list << item if even?(item)
  end
  return_list
end
```

```
list = [1,2,3,4,5,6,7,8,9]
reject_evens(list)
#=> [1,3,5,7,9]
reject_odds(list)
#=> [2,4,6,8]
```

another example

```
def even?(num)
  num % 2 == 0
end
```

```
def reject_evens(list)
  return_list = Array.new
  list.each do |item|
    return_list << item unless even?(item)
  end
  return_list
end
```

```
def reject_odds(list)
  return_list = Array.new
  list.each do |item|
    return_list << item if even?(item)
  end
  return_list
end
```

```
list = [1,2,3,4,5,6,7,8,9]
reject_evens(list)
#=> [1,3,5,7,9]
reject_odds(list)
#=> [2,4,6,8]
```

another example

```
def even?(num)
  num % 2 == 0
end
```

```
def reject_evens(list)
  return_list = Array.new
  list.each do |item|
    return_list << item unless even?(item)
  end
  return_list
end
```

```
def reject_odds(list)
  return_list = Array.new
  list.each do |item|
    return_list << item if even?(item)
  end
  return_list
end
```

```
list = [1,2,3,4,5,6,7,8,9]
reject_evens(list)
#=> [1,3,5,7,9]
reject_odds(list)
#=> [2,4,6,8]
```

```
def reject(list, &block)
  return_list = Array.new
  list.each do item
    return_list << item if block.call(item)
  end
  return_list
end
```

```
list = [1,2,3,4,5,6,7,8,9]
```

```
reject(list) {|i| even?(i) }
```

```
#=> [1,3,5,7,9]
```

```
reject(list) {|i| !even?(i) }
```

```
#=> [2,4,6,8]
```

now we write a general reject function that accepts a block as an argument for the piece that changes, the predicate.

So now we can reject even's and odds using the same function, just by passing in a different block.

```
def make_rejector(&block)
  lambda do |list|
    return_list = Array.new
    list.each do |item|
      return_list << element unless block.call(item)
    end
    return_list
  end
end
```

```
list = [1,2,3,4,5,6,7,8,9]
```

```
odd_rejector = make_rejector { |i| odd?(i) }
odd_rejector.call(list)
#=> [2,4,6,8]
```

make_rejector is exactly the same as reject, except it doesn't return a list.

make_rejector returns a function. The function it returns accepts 1 parameter, a list.

Now that function will include the block passed to make_rejector when first called.

```
def make_rejector(&block)
  lambda do |list|
    return_list = Array.new
    list.each do |item|
      return_list << element unless block.call(item)
    end
    return_list
  end
end
```

```
list = [1,2,3,4,5,6,7,8,9]
```

```
odd_rejector = make_rejector { |i| odd?(i) }
odd_rejector.call(list)
#=> [2,4,6,8]
```

make_rejector is exactly the same as reject, except it doesn't return a list.

make_rejector returns a function. The function it returns accepts 1 parameter, a list.

Now that function will include the block passed to make_rejector when first called.

```
def make_rejector(&block)
  lambda do |list|
    return_list = Array.new
    list.each do |item|
      return_list << element unless block.call(item)
    end
    return_list
  end
end
```

```
list = [1,2,3,4,5,6,7,8,9]
```

```
odd_rejector = make_rejector { |i| odd?(i) }
odd_rejector.call(list)
#=> [2,4,6,8]
```

make_rejector is exactly the same as reject, except it doesn't return a list.

make_rejector returns a function. The function it returns accepts 1 parameter, a list.

Now that function will include the block passed to make_rejector when first called.

Closure

- A Proc that binds locally scoped variables by 'closing' over them
- Captures local variables and keeps them around even after they've gone out of scope

```
def make_rejector(&block)
  lambda do |list|
    return_list = Array.new
    list.each do |item|
      return_list << element if block.call(item)
    end
    return_list
  end
end
```

example closures

```
def complement f
  lambda { |*args| not f.call(*args) }
end
```

```
even? = lambda { |n| n % 2 == 0 }
odd?   = complement(even?)
```

```
odd?.call(1) # true
odd?.call(2) # false
```

complement accepts a function as a parameter and returns a function. The returned function just returns the opposite of the original function passed in.

```
def compose f, g
  lambda { |*args| f.call(g.call(*args)) }
end

find = lambda { |name| User.find_by_username(name) }
auth  = lambda { |u| User.authenticate(u) }
find_and_authenticate = compose(auth, find)

find_and_authenticate.call("Erock") #=> true
```

in the wild

```
def returning(value) #active_support  
  yield(value)  
  value  
end
```

```
returning([]) do |list|  
  list << 1  
  list << 2  
end  
# => [1,2]
```

```
respond_to do |format|
  format.html
  format.js { render :action => "index.rjs" }
  format.xml { render :xml => @user.to_xml }
end
```

```
# Rails RESTful routing
map.resources :users do |user|
  user.resources :blogs
end
```

@user is a local variable to the controller action

```
# Rspec
require 'bowling'

describe Bowling do
  before(:each) do
    @bowling = Bowling.new
  end

  it "should score 0 for gutter game" do
    20.times { @bowling.hit(0) }
    @bowling.score.should == 0
  end
end
```

more info

- SICP: <http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-001Spring-2005/CourseHome/>
- [http://en.wikipedia.org/wiki/Closure_\(computer_science\)](http://en.wikipedia.org/wiki/Closure_(computer_science))
- <http://www.slideshare.net/jashmenn/higher-order-procedures-in-ruby-15799/>
- <http://www.artima.com/intv/closures.html>
- <http://www.randomhacks.net/articles/2007/02/01/some-useful-closures-in-ruby>