# Custom Iterators

# Objective

# We ended on a bit of a cliff hanger in the previous exercise. The last step was to print the total points for each player, like so:

Moe's point totals:  
125 grand total points  
  
Curly's point totals:  
525 grand total points  
  
Larry's point totals:  
275 grand total points

In this exercise we want to expand on that by also printing a breakdown of a player's points on a *per-treasure* basis. Think of it like a receipt of all the treasures a player found during the game and their total point values. Here's an example of the output we're shooting for:

Larry's point totals:  
**200 total skillet points**  
**50 total hammer points**  
**25 total bottle points**  
275 grand total points

But there's a small wrinkle. Consider that the Game class is responsible for printing out this information. But where is this treasure name-total points information currently stored? Well, it's in the @found\_treasures hash in the Player class. So somehow the Game class has to get that information from the Player class.

One approach would be to make the @found\_treasures hash a readable attribute of the Player class. That would allow the Game class to access the hash and iterate over it to print the information. This has a downside, though: it exposes some of the inner workings of the Player class. Specifically, the Game class would rely on the fact that a player's treasures are stored in a hash. And when one class becomes dependent on the implementation of another, it makes changing (maintaining) the program all the more difficult. We call this unnecessary *coupling*.

Instead, we want the Game class to be able to iterate through a player's treasures without revealing how those treasures are stored. To do that, we'll write an iterator method called each\_found\_treasure in the Player class. The Game class can then call this iterator method with an associated block, and the iterator will yield each unique treasure to the block. That way the Game class can do whatever it wants with each treasure. Here's an example of how it'll work:

player.each\_found\_treasure **do** |treasure|  
 puts "#{treasure.points} total #{treasure.name} points"  
**end**

More to the point, the Game class won't know (or care) how the Player class stores those treasures internally. And that's a good thing!

1. Play with Yielding to Blocks

So far when we've used blocks it's been in the context of calling a built-in Ruby method, such as each. Some folks think these built-in iterator methods are in some way special and beyond the reach of most programmers. Thankfully, that's not the case in Ruby. In fact, it's relatively straightforward to write your own methods that call blocks.

Any method can accept a block of code and invoke it by calling yield from within the method. (If you call yield with a parameter, that parameter will be passed as a block parameter to the block.) Then, when the block returns, the method carries on executing any statements after the yield statement. It's that simple.

To get your feet wet with methods that call blocks, try the following examples. We recommend doing this in a new program file called iterators.rb, since multi-line blocks can be cumbersome in irb.

1. Start by writing a method called conversation that takes a block. Here's an example of how you'll call the method with an associated block:
2. conversation { puts "Good to meet you!" }
3. And that would give you the following output:
4. Hello  
   Good to meet you!  
   Goodbye

def conversation  
 puts "Hello"  
 yield  
 puts "Goodbye"  
end

1. Next, write an iterator method called five\_times that invokes a block (wait for it) five times, each time passing the current iteration count. Remember that an *iterator* is just a "normal" method that invokes an associated block of code repeatedly. Here's an example of how you'll call the method:
2. five\_times **do** |n|  
    puts "#{n} situps"  
    puts "#{n} pushups"  
    puts "#{n} chinups"  
   **end**
3. Running that code should give you this output:
4. 1 situps  
   1 pushups  
   1 chinups  
   ...  
   5 situps  
   5 pushups  
   5 chinups
5. Start by doing it with a linear approach where you call yield on five separate lines.

def five\_times  
 yield 1  
 yield 2  
 yield 3  
 yield 4  
 yield 5  
end

1. Then change it to use the upto method to loop from 1 to 5. Each time through the loop, call yield and pass the current iteration count as a parameter to the block.

def five\_times  
 1.upto(5) do |count|  
 yield count  
end  
end

1. Finally, let's try a more generic iterator that doesn't have a hard-coded maximum count. Write a method called n\_timesthat takes a parameter named number. Inside the method, loop from 1 to the value of number. Each time through the loop, call yield and pass the current iteration count as a parameter to the block. Here's an example of how you'll call the method:
2. n\_times(5) **do** |n|  
    puts "#{n} situps"  
    puts "#{n} pushups"  
    puts "#{n} chinups"  
   **end**
3. Running that code should produce the same output as before:
4. 1 situps  
   1 pushups  
   1 chinups  
   ...  
   5 situps  
   5 pushups  
   5 chinups

def n\_times(number)  
 1.upto(number) do |count|  
 yield count  
 end  
end

1. Now the method is more flexible. Try calling it with a different number to iterate more times.

Those are just a few examples of methods yielding to blocks. So the next time you use a built-in method that takes a block, think about how it may be implemented. It's often surprisingly straightforward!

### 2. Yield Treasures to the Game

### Now let's apply what we learned to the game. Returning to the objective, in the Game class we want to be able to iterate through a player's treasures like this:

player.each\_found\_treasure **do** |treasure|  
 puts "#{treasure.points} total #{treasure.name} points"  
**end**

And here's a spec that expresses our objective, which you can go ahead and copy into your player\_spec.rb spec file:

it "yields each found treasure and its total points" **do**  
 @player.found\_treasure(Treasure.new(**:skillet**, 100))  
 @player.found\_treasure(Treasure.new(**:skillet**, 100))  
 @player.found\_treasure(Treasure.new(**:hammer**, 50))  
 @player.found\_treasure(Treasure.new(**:bottle**, 5))  
 @player.found\_treasure(Treasure.new(**:bottle**, 5))  
 @player.found\_treasure(Treasure.new(**:bottle**, 5))  
 @player.found\_treasure(Treasure.new(**:bottle**, 5))  
 @player.found\_treasure(Treasure.new(**:bottle**, 5))  
  
 yielded = []  
 @player.each\_found\_treasure **do** |treasure|  
 yielded << treasure  
 **end**  
  
 yielded.should == [  
 Treasure.new(**:skillet**, 200),  
 Treasure.new(**:hammer**, 50),  
 Treasure.new(**:bottle**, 25)  
 ]  
**end**

Notice here that we call each\_found\_treasure with a block. Inside the block we add each treasure the iterator gives us to the yielded array. That way we can specify that we expect three treasures to end up in the yielded array. Why only three treasures? Because remember that the player stores a unique list of treasures, where each treasure has a name and the *total*point value.

Given what you've learned, this task is well within your reach...

1. In the Player class, define a method called each\_found\_treasure that iterates over the @found\_treasures hash. For each key-value pair in the hash, create a new Treasure object (it's a Struct) whose name is the same as the key in the @found\_treasures hash and whose points is the same as the value in the @found\_treasures hash. Yield this representative Treasure object to the associated block. In order for the Player class to use the Treasure class, you'll need to require the treasure\_trove.rb file.

require\_relative 'treasure\_trove'

class Player  
 def each\_found\_treasure  
 @found\_treasures.each do |name, points|  
 yield Treasure.new(name, points)  
 end  
end  
end

1. Run the player\_spec.rb spec file to make sure it passes.
2. All that's left to do is use the iterator in the game. Remember, we want to print a breakdown of each player's points on a per-treasure basis, like this:
3. Larry's point totals:  
   200 total skillet points  
   50 total hammer points  
   25 total bottle points  
   275 grand total points
4. Inside the print\_stats method of the Game class, iterate through each player in the game. For each player, call theeach\_found\_treasure method with a block that takes a treasure as a block parameter. Inside the block, print out the treasure's name and points.

@players.sort.each do |player|  
 puts "\n#{player.name}'s point totals:"  
 player.each\_found\_treasure do |treasure|  
 puts "#{treasure.points} total #{treasure.name} points"  
 end  
 puts "#{player.points} grand total points"  
end

### Solution

The full solution for this exercise is in the iterators directory of the [code bundle](https://s3.amazonaws.com/pragmaticstudio/courses/ruby/pragstudio-ruby-code.zip).

### Bonus Round

#### An Alternative Ending

Here's a little bonus game feature to stretch your block yielding skills...

Suppose we want to attach an arbitrary block of code to the game's play method and invoke that block before the start of every round. If the block returns true, then the game should end. Otherwise, the next round should be played. For example, arrange things so that the game automatically ends when the game exceeds 2,000 total points. In other words, make the following code work:

knuckleheads.play(10) **do**  
 knuckleheads.total\_points >= 2000  
**end**

Remember that the value of the last expression evaluated in a block is passed back to the method as the value of the yield statement. So if the result of invoking the block returns true, you want to break out of the play loop. To do that, you'll need to use the break keyword.

1.upto(rounds) do |round|  
 if block\_given?  
 break if yield  
 end  
end

Try running the game with 10 rounds or more and it should stop before all the rounds are played.

The benefit of using a block here is that the decision of whether to end the game can vary depending on how the play method is called. As the developer who created the Game class, you don't have to make that decision. Rather, the developer who uses the Game class gets to make that decision without prying into the inner workings of the Game class.

In short, we've *decoupled* the concept of playing through rounds of a game from the logic that determines when the game ends. And that makes our code more flexible in the long run.

#### Fundraising Program

Want some more practice with custom iterators? Here's an idea: In your fundraising program, print out a project's pledges on a per-pledge basis. Think of it as a way to show off how much a particular project received in gold pledges. The output could be something such as:

Project ABC's pledges:  
$50 in bronze pledges  
$150 in silver pledges  
$500 in gold pledges  
$700 in total pledges

fund\_request.rb

require\_relative 'project'

require\_relative 'funding\_round'

require\_relative 'pledge\_pool'

class FundRequest

attr\_reader :title

def initialize(title)

@title = title

@projects = []

end

def add\_project(a\_project)

@projects.push(a\_project)

end

def request\_funding(rounds)

puts "There are #{@projects.size} projects that you could fund:"

@projects.each do |project|

puts project

end

pledges = PledgePool::PLEDGES

puts "\nThere are #{pledges.size} possible pledge amounts:"

pledges.each do |pledge|

puts "A #{pledge.name} pledge is worth $#{pledge.amount}."

end

1.upto(rounds) do |round|

puts "\nFunding Round #{round}:"

@projects.each do |project|

FundingRound.one\_round(project)

puts project

end

end

end

def print\_name(project)

puts "#{project.name}"

end

def print\_results

fully\_funded\_projects = @projects.select { |project| project.fully\_funded? }

under\_funded\_projects = @projects.reject { |project| project.fully\_funded? }

puts "\n#{fully\_funded\_projects.size} Fully Funded Projects:"

fully\_funded\_projects.each do |project|

print\_name(project)

end

puts "\n#{under\_funded\_projects.size} Under Funded Projects:"

under\_funded\_projects.each do |project|

print\_name(project)

end

sorted\_projects = under\_funded\_projects.sort { |a, b| b.total\_funding\_outstanding <=> a.total\_funding\_outstanding}

puts "\n#{under\_funded\_projects.size} projects still need your help:"

sorted\_projects.each do |project|

formatted\_name = project.name.ljust(20, '.')

puts "#{formatted\_name} $#{project.total\_funding\_outstanding} under"

end

@projects.each do |project|

puts "\n#{project.name}'s pledges:"

project.each\_received\_pledge do |pledge|

puts "$#{pledge.amount} in #{pledge.name} pledges"

end

puts "$#{project.pledges} in total pledges"

end

end

end

fund\_request\_specifications.rb

require\_relative 'fundrequest'

describe FundRequest do

before do

@fundrequest = FundRequest.new("VC-Friendly Start-up Projects")

@initial\_funds = 1000

@project = Project.new("Project ABC", 5000, @initial\_funds)

@fundrequest.add\_project(@project)

end

it "adds funds to a project if an even number is rolled" do

Die.any\_instance.stub(:roll).and\_return(4)

@fundrequest.request\_funding(2)

@project.funding.should == @initial\_funds + (25 \* 2)

end

it "removes funds to a project if an odd number is rolled" do

Die.any\_instance.stub(:roll).and\_return(3)

@fundrequest.request\_funding(2)

@project.funding.should == @initial\_funds - (15 \* 2)

end

it "assigns a pledge for amount during a project's funding round" do

fundrequest = FundRequest.new("VC-Friendly Start-up Projects")

project = Project.new("Project ABC", 5000)

fundrequest.add\_project(project)

fundrequest.request\_funding(1)

project.pledges.should\_not be\_zero

end

end

pledge\_pool.rb

Pledge = Struct.new(:name, :amount)

module PledgePool

PLEDGES = [

Pledge.new(:bronze, 50),

Pledge.new(:silver, 75),

Pledge.new(:gold, 100)

]

def self.random

PLEDGES.sample

end

end

pledge\_pool\_specifications.rb

require\_relative 'pledge\_pool'

describe Pledge do

before do

@pledge = Pledge.new(:bronze, 50)

end

it "has a name attribute" do

@pledge.name.should == :bronze

end

it "has an amount attribute" do

@pledge.amount.should == 50

end

end

describe PledgePool do

it "has three pledges" do

PledgePool::PLEDGES.size.should == 3

end

it "has a bronze pledge worth $50" do

PledgePool::PLEDGES[0].should == Pledge.new(:bronze, 50)

end

it "has a silver pledge worth 75 points" do

PledgePool::PLEDGES[1].should == Pledge.new(:silver, 75)

end

it "has a gold pledge worth 100 points" do

PledgePool::PLEDGES[2].should == Pledge.new(:gold, 100)

end

it "returns a random pledge" do

pledge = PledgePool.random

PledgePool::PLEDGES.should include(pledge)

end

end

project.rb

require\_relative 'pledge\_pool'

class Project

attr\_accessor :name

attr\_reader :funding, :target

def initialize(name, target\_funding\_amount, funding=0)

@name = name

@target = target\_funding\_amount

@funding = funding

@received\_pledge = Hash.new(0)

end

def to\_s

"#{@name} has $#{total\_funds} in funding towards a goal of $#{@target}."

end

def remove\_funds

@funding -= 15

puts "#{@name} lost some funds!"

end

def add\_funds

@funding += 25

puts "#{@name} got more funds!"

end

def total\_funding\_outstanding

@target - total\_funds

end

def fully\_funded?

total\_funding\_outstanding <= 0

end

def received\_pledge(pledge)

@received\_pledge[pledge.name] += pledge.amount

puts "#{@name} received a #{pledge.name} pledge worth $#{pledge.amount}."

puts "#{@name}'s pledges: #{@received\_pledge}"

end

def pledges

@received\_pledge.values.reduce(0, :+)

end

def total\_funds

@funding + pledges

end

def each\_received\_pledge

@received\_pledge.each do |name, amount|

yield Pledge.new(name, amount)

end

end

end

if \_\_FILE\_\_ == $0

project = Project.new("Project ABC", 5000, 1000)

puts project.name

puts project.funding

project.remove\_funds

puts project.funding

project.add\_funds

puts project.funding

end

project\_specifications.rb

require\_relative 'project'

describe Project do

before do

@initial\_funds = 1000

@project = Project.new("Project ABC", 5000, @initial\_funds)

$stdout = StringIO.new

end

it "has an initial target funding amount" do

@project.target.should == 5000

end

it "computes the total funds outstanding as the target funding amount minus the funding amount" do

@project.total\_funding\_outstanding.should == (5000 - 1000)

end

it "increases funds by 25 when funds are added" do

@project.add\_funds

@project.funding.should == @initial\_funds + 25

end

it "decreases funds by 15 when funds are removed" do

@project.remove\_funds

@project.funding.should == @initial\_funds - 15

end

context "created without a funding amount" do

before do

@project = Project.new("ProjectABC", 5000)

end

it "has a default funding amount of 0" do

@project.funding.should == 0

end

end

context "when total funding outstanding is less than or equal to 0" do

before do

@project = Project.new("ProjectABC", 5000, 5000)

end

it "is fully-funded" do

@project.should be\_fully\_funded

end

end

context "when total funding outstanding is greater than 0" do

before do

@project = Project.new("ProjectABC", 5000, 1000)

end

it "is under-funded" do

@project.should\_not be\_fully\_funded

end

end

it "computes pledges as the sum of all pledges" do

@project.pledges.should == 0

@project.received\_pledge(Pledge.new(:silver, 75))

@project.pledges.should == 75

@project.received\_pledge(Pledge.new(:gold, 100))

@project.pledges.should == 175

@project.received\_pledge(Pledge.new(:gold, 100))

@project.pledges.should == 275

end

it "computes total funds as the sum of a projects funding and pledges" do

@project.received\_pledge(Pledge.new(:gold, 100))

@project.received\_pledge(Pledge.new(:gold, 100))

@project.total\_funds.should == 1200

end

it "yields each received pledge and its total pledge amount" do

@project.received\_pledge(Pledge.new(:bronze, 50))

@project.received\_pledge(Pledge.new(:silver, 75))

@project.received\_pledge(Pledge.new(:silver, 75))

@project.received\_pledge(Pledge.new(:gold, 100))

@project.received\_pledge(Pledge.new(:gold, 100))

@project.received\_pledge(Pledge.new(:gold, 100))

yielded = []

@project.each\_received\_pledge do |pledge|

yielded << pledge

end

yielded.should == [

Pledge.new(:bronze, 50),

Pledge.new(:silver, 150),

Pledge.new(:gold, 300)

]

end

end

crowdfund.rb

require\_relative 'project'

require\_relative 'fundrequest'

project1 = Project.new("Project ABC", 5000, 1000)

project2 = Project.new("Project LMN", 3000, 500)

project3 = Project.new("Project XYZ", 75, 25)

projects = FundRequest.new("VC-Friendly Start-up Projects")

puts projects.title

projects.add\_project(project1)

projects.add\_project(project2)

projects.add\_project(project3)

projects.request\_funding(3)

projects.print\_results

die.rb

class Die

attr\_reader :number

def initialize

roll

end

def roll

@number = rand(1..6)

end

end

if \_\_FILE\_\_ == $0

die = Die.new

puts die.roll

puts die.roll

puts die.roll

end

funding\_round.rb

require\_relative 'project'

require\_relative 'die'

require\_relative 'pledge\_pool'

module FundingRound

def self.one\_round(project)

die = Die.new

number\_rolled = die.roll

if number\_rolled.odd?

project.remove\_funds

else

project.add\_funds

end

pledge = PledgePool.random

project.received\_pledge(pledge)

end

end

### Wrap Up

As a takeaway, we hope this exercise gave you a glimpse into how blocks offer a different style of programming that can inform the design of your program. Indeed, by understanding how to write your own iterators, you'll have another design technique in your repertoire. And you might just find that blocks and iterators make your code easier to read and maintain.

This exercise may also have revealed some of the magic of Ruby. Sorry 'bout that. The reality is, there isn't any magic. And once you know how things work in Ruby you immediately start to feel a lot more confident.

In the next section we'll explore how to handle input and output in Ruby: entering the number of rounds of play from the command line, loading players from a file, and saving high scores to a file. It's all coming up!