At this point of the course, you are certainly comfortable using operations like || and && for boolean values, but now let's explore how these operations can be used on other data types.

### Truthy and Falsey Values

Our current understanding of || applies to the boolean values of true and false, so how can we use those operations on non-boolean values? It is the case that every value in Ruby can be treated as true (truthy) or as false (falsey). Remembering which values are truthy and which are falsey in Ruby is simple:

* nil and false are the only falsey values
* everything else is truthy

Note that these rules for truthy vs falsey are specific to Ruby. Other languages may differ in their truthy and falsey values. For example, in JavaScript, the number 0 and the empty string "" are considered falsey as well. This is not the case in Ruby.

The implication of this is that we can use nil to act as false in scenarios that traditionally require the boolean:

val = nil

if val

p "it is true"

else

p "it is false"

end

# The code above will print "it is false" since nil is a falsey value

### What does logical OR really do?

So far in the course, we've been thinking about how || works in a somewhat naive way. You probably agree with following description of the expression a || b, where a and b are any values :

* when at least one value is true, we return true
* when both sides are false, we return false

While "correct", this does not fully describe || in a way that shows it's full potential. Here is a more complete description of how a || b behaves under the hood:

* when a is truthy, return a
* when a is falsey, return b

With this second description in mind, we can use || on any values:

true || 42 # => true

42 || true # => 42

false || 42 # => 42

42 || false # => 42

false || "hello" # => "hello"

nil || "hello" # => "hello"

"hi" || "hello" # => "hi"

0 || true # => 0

false || nil # => nil

### Default Arguments

Using the evaluation of || in this way is particularly useful when implementing default arguments for our methods. Take the following code as an example:

def greet(person = nil)

if person.nil?

person = "you"

end

p "Hey " + person

end

greet("Brian") # => "Hey Brian"

greet # => "Hey you"

We can refactor the conditional by utilizing || and the fact that nil is a falsey value:

def greet(person = nil)

person = person || "you"

p "Hey " + person

end

greet("Brian") # => "Hey Brian"

greet # => "Hey you"

But wait, there's more! We can utilize our classic shorthand. We commonly write a += b in place of a = a + b. In the same way, we can write a ||= b in place of a = a || b :

def greet(person = nil)

person ||= "you"

p "Hey " + person

end

greet("Brian") # => "Hey Brian"

greet # => "Hey you"

You're probably wondering why we don't simply assign person = "you" directly in the parameter list for the method. You are right! We should to make the code as succinct and to the point as possible using that syntax. We'll admit that this was an exercise in getting comfortable utilizing nil as a falsey value for use with ||. But now that we are comfortable with this logic, we can implement default procs up next!

### Default Procs

The ||= pattern is utilized heavily when implementing default procs:

def call\_that\_proc(val, &prc)

prc ||= Proc.new { |data| data.upcase + "!!" }

prc.call(val)

end

p call\_that\_proc("hey") # => "HEY!!"

p call\_that\_proc("programmers") { |data| data \* 3 } # => "programmersprogrammersprogrammers"

p call\_that\_proc("code") { |data| "--" + data.capitalize + "--"} # => "--Code--"

You'll notice that in the above code, we don't explicitly assign prc to be nil. This is because prc will automatically contain nil if the method is called without passing in a proc.

### Lazy Initialization

The ||= pattern is also useful when implementing Lazy Initialization for classes. Lazy initialization is a design strategy where we delay creation of an object until it is needed. The idea is to avoid slow or costly operations until they are absolutely necessary. This contrasts with our typical classes that preemptively set all attributes up front. For example, take this Restaurant class that initializes all attributes immediately:

class Restaurant

attr\_accessor :name, :chefs, :menu

def initialize(name, chefs)

@name = name

@chefs = chefs

@menu = ["sammies", "big ol' cookies", "bean blankies", "chicky catch", "super water"]

end

end

five\_star\_restaurant = Restaurant.new("Appetizer Academy", ["Marta", "Jon", "Soon-Mi"])

p five\_star\_restaurant

#<Restaurant:0x00007fea7a8c6880

# @name="Appetizer Academy",

# @chefs=["Marta", "Jon", "Soon-Mi"],

# @menu=["sammies", "big ol' cookies", "bean blankies", "chicky catch", "super water"]

#>

While it is required that @name and @chefs must be assigned immediately in Restaurant#initialize since they are taken in as arguments, it is not necessary that @menu be assigned immediately. Imagine that @menu was a "costly" object like an array of 10,000 elements. Initializing @menu may slow down the creation of the Restaurant. To overcome this, we'll use the lazy initialization pattern to only create the @menu if someone asks for it. In other words, we'll create the @menu in the Restaurant#menu getter if it does not exist already:

class Restaurant

attr\_accessor :name, :chefs, :menu

def initialize(name, chefs)

@name = name

@chefs = chefs

end

def menu

@menu ||= ["sammies", "big ol' cookies", "bean blankies", "chicky catch", "super water"]

end

end

five\_star\_restaurant = Restaurant.new("Appetizer Academy", ["Marta", "Jon", "Soon-Mi"])

p five\_star\_restaurant

#<Restaurant:0x00007f90b3922368

# @name="Appetizer Academy",

# @chefs=["Marta", "Jon", "Soon-Mi"]

#>

p five\_star\_restaurant.menu

#["sammies", "big ol' cookies", "bean blankies", "chicky catch", "super water"]

p five\_star\_restaurant

#<Restaurant:0x00007f90b3922368

# @name="Appetizer Academy",

# @chefs=["Marta", "Jon", "Soon-Mi"],

# @menu=["sammies", "big ol' cookies", "bean blankies", "chicky catch", "super water"]

#>

Above, notice how the restaurant lacks a @menu until we call the getter! To accomplish this we leveraged the fact that a missing attribute will be nil. That means we can use the ||= pattern!