**Syntactic Sugar**

As you've been learning with us through the course, you may have noticed we often write code in a particular way only to replace that code style with a shortcut that is more elegant and preferred later on. We introduce the longhand ways of writing things first so you have a deep understanding of the underlying concepts. Once you do learn the shortcuts, you also have an appreciation of how expressive of a language Ruby is! There's a *method* to the madness.†

The previous attr methods are a good example of this. The broad name for code or *syntax* that is a "shortcut" for other code is "Syntactic Sugar". For example, attr\_reader is syntactic sugar for defining a full getter method! We use syntactic sugar to sweeten up our code, making it more readable.

Let's explore some syntactic sugar methods we can use in our classes:

**Implementing "operator" Methods**

Like we alluded to in previous sections of the course. You can implement +, ==, >, etc.. methods on your classes. Let's say we had a Person class and we wanted to implement equality, ==, on the basis of last\_name:

class Person

attr\_reader :first\_name, :last\_name

def initialize(first\_name, last\_name, age)

@first\_name = first\_name

@last\_name = last\_name

@age = age

end

def ==(other\_person)

self.last\_name == other\_person.last\_name

end

end

person\_1 = Person.new("Jane", "Doe", 20)

person\_2 = Person.new("John", "Doe", 18)

person\_3 = Person.new("John", "Wayne", 18)

# Calling Person#== without any syntactic sugar is awkward:

p person\_1.==(person\_2) # true

# With syntactic sugar, it's much more elegant:

p person\_1 == person\_2 # true

p person\_2 == person\_3 # false

As always, if you treat Person#== as a method where "==" is the name, then we can call it with person\_1.==(person\_2). In this context since person\_1 is the object calling ==, self will refer to person\_1 inside of the method. person\_2 is obviously the other\_person argument since it is being passed as an arg within the parens. Syntactic sugar comes in when we drop the . and (). The resulting calls are much cleaner!

For these methods like ==, +, >, etc. you can typically identify the object to the left as what self will refer to within the method. The argument will be the object to the right. This is exactly the case in our Person#== method.

**Implementing Bracket Methods**

Let's explore how to implement a custom #[] method in our classes. We often use this method to index an Array or key a Hash.

Say we wanted to have a Queue#[] method to return the element at the given position in the @line. Like we explored in our abstraction and encapsulation lesson, we want to avoid giving direct access to the @line array.

class Queue

def initialize

@line = []

end

def [](position)

@line[position]

end

def add(ele)

@line << ele # add ele to back of line

nil

end

def remove

@line.shift # remove front ele of line

end

end

grocery\_checkout = Queue.new

grocery\_checkout.add("Alan")

grocery\_checkout.add("Alonzo")

# Calling Queue#[] without any syntactic sugar is ugly:

grocery\_checkout.[](0) # "Alan"

grocery\_checkout.[](1) # "Alonzo"

# With syntactic sugar, it's waaaaay better:

grocery\_checkout[0] # "Alan"

grocery\_checkout[1] # "Alonzo"

Notice that the syntactic sugar when calling the #[] method is more involved than the previous examples. grocery\_checkout.[](1) is equivalent to grocery\_checkout[1]. Using the syntactic sugar, we can put a number between the square brackets and it will be interpreted as the first arg to the #[] method.

**Implementing #[]=**

Building off of the #[] method what if I wanted the ability to assign to a specified position by implementing a #[]= method? Let's take a look at how we can define this special setter:

class Queue

def initialize

@line = []

end

def [](position)

@line[position]

end

def []=(position, ele)

@line[position] = ele

end

def add(ele)

@line << ele # add ele to back of line

nil

end

def remove

@line.shift # remove front ele of line

end

end

grocery\_checkout = Queue.new

grocery\_checkout.add("Alan")

grocery\_checkout.add("Alonzo")

# Let's call Queue#[]= without syntactic sugar:

grocery\_checkout.[]=(0, "Grace")

p grocery\_checkout #<Queue:0x007fe7a7a29ec8 @line=["Grace", "Alonzo"]>

# Let's call Queue#[]= again, but with syntactic sugar:

grocery\_checkout[1] = "Grace"

p grocery\_checkout #<Queue:0x007fe7a7a29ec8 @line=["Grace", "Grace"]>

From the above we see that grocery\_checkout.[]=(0, "Grace") is equivalent to grocery\_checkout[0] = "Grace". Looking at how we defined the []= method, we can gather that when we use the syntactic sugar for calling []=, the data that we put between the brackets will be the first argument, and what comes after the equal sign will be the second argument to our method

† pun intended