## Class Basics

### Motivation

Before we learn about classes, let's get motivated! Why should we use classes and what advantages do they offer?

#### How to Create Cats Poorly

As a programmer, you'll often want to model some object and the properties of that object. For example, a social media site may need to model a User with their username and a profile picture. Or perhaps a music site may need to model a Song with its title, genre, and duration. Following App Academy tradition, let's say we wanted to model some Cats in ruby! Our Cats will have names, colors, and ages:

cat\_1 = {name: "Sennacy", color: "brown", age: 3}

cat\_2 = {name: "Whiskers", color: "white", age: 5}

cat\_3 = {name: "Garfield", color: "orange", age: 7}

Above we used hashes to represent our Cats. This seems like a fair choice, because we can use the key:value pairs of hashes to represent the properties of our Cats. For example cat\_1 has a name of "Sennacy" and an color of "brown". Now, imagine we wanted to create a thousand Cats. We would have to tediously create each hash with the same keys of name, color, and age. This leaves a lot of opportunity for typos. We want to follow the DRY rule (**D**on't **R**epeat **Y**ourself) and improve this code. By using a class we can avoid this repetition and easily create objects of the same structure.

### Creating a Cat Class

In the cat example above, we can notice a theme across all Cats we create. They all have the same keys, but may differ in their values. We can say that each Cat has the same structure. Let's DRY up the last example by creating a Class to act as a blueprint for Cats.

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

end

A few things we'll want to note about the code above:

* to create a class we use the class keyword, big surprise
* the name of a class must begin with a capital letter
* we can define methods within a class

You'll notice that we defined a method named initialize in our class. This is a special method name that we will use when creating cats. The method expects 3 parameters, which is nothing new, but what are @name, @color, etc.? @ is how we denote a instance variable or attribute of our class. That means that our cats will have the attributes or properties of @name, @color, @age.

#### Initializing New Cats

Now that we have a Cat class, we have a blueprint that can easily create Cats for us. Let's put it to use:

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

end

cat\_1 = Cat.new("Sennacy", "brown", 3)

cat\_2 = Cat.new("Whiskers", "white", 5)

p cat\_1 #<Cat:0x007fb6d804cfe0 @age=3, @color="brown", @name="Sennacy">

p cat\_2 #<Cat:0x007fb6d6bb60b8 @age=5, @color="white", @name="Whiskers">

Let's recognize something a bit strange about this code: To create a Cat we must call Cat.new, this must mean that new is a method on Cat. This is strange because nowhere did we define a method literally named new. The trick is, when we call Cat.new, ruby will be really calling upon the initialize method we defined. A hint at this is the fact that the initialize method expects a name, color, age and when we call Cat.new we pass in a name, color, age. You're probably wondering why the heck we can't just call Cat.initialize; it seems way more logical right??? The short answer to that is because reasons. This is something we'll have to accept blindly for now: Cat.new will execute our initialize method. As we explore more about classes we promise to explain the weirdness behind new and initialize.

With that out of the way, let's get to the punchline. When we call Cat.new("Sennacy", "brown", 3), it will return an object to us that we store in the variable cat\_1. Notice that the object contains the attributes that result from executing initialize. If we want to create more cats we simply call Cat.new again, passing in any name, color, age we please. We can use our Cat class to create any number of Cat instances. This means that cat\_1 and cat\_2 are instances of Cat.

Notice that when we print out an instance of a class, the notation will show which class this instance belongs to and a unique id for this object: <Cat:0x007fb6d804cfe0...

#### Getter Methods

Since we designed a Cat instance to consist of 3 attributes, it's common to also want a way to refer to the value of those attributes. To do this, we define "Getter Methods" to get (return) those attributes. Let's add a name getter to Cat:

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

def get\_name

@name

end

end

cat\_1 = Cat.new("Sennacy", "brown", 3)

p cat\_1.get\_name # "Sennacy"

Notice that we defined another method called get\_name in our class. To call this method, we must call it on a Cat instance, not directly on the Cat class! This makes sense because cat\_1 is an instance, so it refers to a particular cat. If we had done Cat.get\_name we would be incorrectly trying to get the name of the blueprint. Cat is just the blueprint, so it does not refer to any single, particular cat. In summary we should call cat\_1.get\_name and not Cat.get\_name.

By convention, getter methods typically have the same name as the attribute they are returning. So instead of defining get\_name, we'll simply define name. Let's add another getter using this convention:

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

def name

@name

end

def age

@age

end

end

cat\_1 = Cat.new("Sennacy", "brown", 3)

p cat\_1.name # "Sennacy"

p cat\_1.age # 3

cat\_2 = Cat.new("Whiskers", "white", 5)

p cat\_2.name # "Whiskers"

p cat\_2.age # 5

p cat\_2.color # This will give NoMethodError: undefined method `color'

Cool, so we can now refer to the name and age of any Cat instance! Note that if we don't create a getter for a particular attribute, we won't have a way to refer to that attribute. Such as in the example above, we cannot refer to a Cat's color because we did not create the corresponding getter.

A final thought about getter methods, because they simply return the value of an attribute, we cannot use them to modify the @attribute. So we cannot use a getter method to change a cat's age.

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

def name

@name

end

end

cat\_1 = Cat.new("Sennacy", "brown", 3)

p cat\_1.name # "Sennacy"

cat\_1.name = "Kitty" # This will give NoMethodError: undefined method `name='

To do accomplish this behavior we'll need to learn about setter methods next!

#### Setter Methods

Let's see what happens when we try to assign to an attribute of a Cat instance without the proper method in place. The following code will not work:

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

# getter

def age

@age

end

end

cat\_1 = Cat.new("Sennacy", "brown", 3)

cat\_1.age = 42 # NoMethodError: undefined method `age='

The error we get above suggests that we need to have a age= method on our Cat class. What a strange method name! Let's implement it:

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

# getter

def age

@age

end

# setter

def age=(number)

@age = number

end

end

cat\_1 = Cat.new("Sennacy", "brown", 3)

p cat\_1 #<Cat:0x007f8511a6f340 @age=3, @color="brown", @name="Sennacy">

cat\_1.age = 42

p cat\_1 #<Cat:0x007f8511a6f340 @age=42, @color="brown", @name="Sennacy">

Now we have a working method that we can use to change the age! Great. But something that feels uncomfortable here is how we call the method with cat\_1.age = 42. If age= is the method name, then what's up with the space between age and =, as well as the lack of parentheses around our 42 arg? This doesn't seem like a proper method call, but it truly is! The following two method calls are equivalent

cat\_1.age=(42)

cat\_1.age = 42

For setter methods especially, we'll prefer the second version because the syntax is cleaner. Ruby is a quite flexible language. In general you are not required to use parentheses around arguments when making a method call. Try it for yourself: "aeiou".include?("e") is equivalent to "aeiou".include? "e". As a matter of style and convention, we'll only omit parentheses for method calls that don't take in args or are special exceptions like a classic setter method.

### Beyond Getters and Setters

Getters and setters are common methods to implement on a class, but we can implement any arbitrary method we please on a class. The possibilities are endless:

class Cat

def initialize(name, color, age)

@name = name

@color = color

@age = age

end

def purr

if @age > 5

puts @name.upcase + " goes purrrrrr..."

else

puts "..."

end

end

end

cat\_1 = Cat.new("Sennacy", "brown", 10)

cat\_1.purr # "SENNACY goes purrrrrr..."

cat\_2 = Cat.new("Whiskers", "white", 3)

cat\_2.purr # "..."

**Instance vs Class Variables**

In our exploration of classes so far we have used plenty of **instance variables** or **attributes**. Similar to how we can have *class methods* we can also have *class variables*. Let's compare the two.

**Instance Variables**

Like we learned previously, instance variables are denoted with @ and are typically assigned inside #initialize:

class Car

def initialize(color)

@color = color

end

def color

@color

end

end

car\_1 = Car.new("red")

p car\_1.color # "red"

car\_2 = Car.new("black")

p car\_2.color # "black"

Nothing new here. If we want cars to vary in the property of color, then we simply make the relevant instance variable for @color. Great, but what if we wanted to have a property that is shared among all cars? Let's accomplish this next using a class variable.

**Class Variables**

Let's say we wanted all car instances to have the same number of wheels. We can add a class variable @@num\_wheels:

class Car

@@num\_wheels = 4

def initialize(color)

@color = color

end

# getter for @color instance variable

def color

@color

end

# getter for @@num\_wheels class variable

def num\_wheels

@@num\_wheels

end

end

car\_1 = Car.new("red")

p car\_1.num\_wheels # 4

car\_2 = Car.new("black")

p car\_2.num\_wheels # 4

Notice that we use @@ to denote class variables and typically assign these variables right inside of the class, but *not* inside of #initialize. This means that any car instance we create will be able to refer to this single, shared @@num\_wheels variable. An important distinction to have in mind is that instances car\_1 and car\_2 have their own/separate @color variables, but share a single @@num\_wheels variable.

As a result of all instances sharing this single variable, a change to this variable will effect all instances. Let's create a class method that sets @@num\_wheels:

class Car

@@num\_wheels = 4

def self.upgrade\_to\_flying\_cars

@@num\_wheels = 0

end

def initialize(color)

@color = color

end

def num\_wheels

@@num\_wheels

end

end

car\_1 = Car.new("red")

car\_2 = Car.new("black")

p car\_1.num\_wheels # 4

p car\_2.num\_wheels # 4

Car.upgrade\_to\_flying\_cars

p car\_1.num\_wheels # 0

p car\_2.num\_wheels # 0

car\_3 = Car.new("silver")

p car\_3.num\_wheels # 0

The future is now! Changing class variables is really powerful since it effects every instance that we created and *will create in the future* (see car\_3 above). However, with great power comes great responsibility, so be very cautious when writing such code.

**Class Constants**

Often times, we'll want to prevent class variables from being changed for safety. In this scenario we'll want to create a **class constant** instead. As it's name suggests, a constant cannot be reassigned. Let's redo the last example with a class constant:

class Car

NUM\_WHEELS = 4

def self.upgrade\_to\_flying\_cars

NUM\_WHEELS = 0 # SyntaxError: dynamic constant assignment

end

def initialize(color)

@color = color

end

def num\_wheels

NUM\_WHEELS

end

end

car\_1 = Car.new("red")

car\_2 = Car.new("black")

p car\_1.num\_wheels # 4

p car\_2.num\_wheels # 4

Car.upgrade\_to\_flying\_cars

Class constant names must be capitalized. Notice that reassigning the constant will fail with an error, exactly what we wanted!

**Wrapping Up**

* an @instance\_variable will be a distinct variable in each instance of a class; changing the variable will only effect that one instance
* a @@class\_variable will be shared among all instances of a class; changing the variable will effect all instances because all instances of the class
* a CLASS\_CONSTANT will be shared among all instances of a class, but cannot be changed

**Instance Methods vs Class Methods**

Now that we have the basics of classes down, let's explore two different methods we may build into class: **instance methods** and **class methods**.

**Instance Methods**

So far we've been only dealing with instance methods with our classes. Like it's name suggests, an instance method is one that is called on an *instance* of a class. Let's check out an instance method:

class Dog

def initialize(name, bark)

@name = name

@bark = bark

end

def speak

@name + " says " + @bark

end

end

my\_dog = Dog.new("Fido", "woof")

my\_dog.speak # "Fido says woof"

other\_dog = Dog.new("Doge", "much bork")

other\_dog.speak # "Doge says much bork"

speak is an instance method because we can only call it on a Dog instance we initialized using Dog.new. Remember that if something is an instance of Dog, it is an object with a @name and @bark. Since my\_dog and other\_dog are instances, when we call speak on them respectively, we can get different behavior because they can have different @name and @bark values. An instance method depends on the *attributes* or *instance variables* of an instance.

For notation, we'll use **Dog#speak** to denote that speak is an **instance method** of Dog

**Class Methods**

A class method is a method that is called directly on the class. Let's see how to define a class method:

class Dog

def initialize(name, bark)

@name = name

@bark = bark

end

def self.growl

"Grrrrr"

end

end

Dog.growl # Grrrrr

Notice that we define class method by adding self. to the front of a method name. In this context, self refers to the Dog class itself. Since growl is a class method, we cannot call it on an instance; instead we call it on the Dog class directly . A class method cannot refer to any instance attributes like @name and @bark! As programmers, we'll choose to build class methods for added utility.

For notation we'll use **Dog::growl** to denote that growl is an **class method** of Dog.

For example, here is a class method that is a bit more practical, Dog::whos\_louder :

class Dog

def initialize(name, bark)

@name = name

@bark = bark

end

def self.whos\_louder(dog\_1, dog\_2)

if dog\_1.bark.length > dog\_2.bark.length

return dog\_1.name

elsif dog\_1.bark.length < dog\_2.bark.length

return dog\_2.name

else

return nil

end

end

def name

@name

end

def bark

@bark

end

end

d1 = Dog.new("Fido", "woof")

d2 = Dog.new("Doge", "much bork")

p Dog.whos\_louder(d1, d2) # "Doge"

You may be wondering why we prefer to make Dog::whos\_louder a class method. We make this choice because the code inside of the method does not pertain to a single instance of dog, meaning it does not refer to instance attributes of @name, @bark.

**Wrapping Up**

* Class#method\_name means method\_name is an instance method
* Class::method\_name means method\_name is a class method