Jade's Guide to Object Oriented Programming

Jade Singh, CS 61A Spring 2020

What is the point of OOP?

Sometimes, when coding, we want to create objects that are more complicated than our basic types (integers, strings, booleans, etc.). We saw that with abstract data types (ADTs), which had constructors that we used to create more complex data types. However, there was an issue with ADTs: some ADTs (like our tree ADT) didn't have selectors that allowed us to mutate our ADT. We were kind of stuck with whatever we got.

Our solution to this problem is **object oriented programming** (abbreviated as OOP). Rather than using ADTs or defining a bunch of complex objects from scratch, we can create what is called a **class**, which is sort of like a template for a particular object. Every time we want to create one of these objects, we can create what is called an **instance** of the class. This prevents us from having to write a lot of redundant code. Every class contains an `__init__` method (see more about the `__init__` method below), which is what is called whenever we create a new instance of a particular class, which we can do by writing the class name followed by parentheses.

```
class Car:
    num_wheels = 4
    def __init__(self, color, make):
        self.color = color
        self.make = make
    def drive(self):
        return 'vroom vroom'
    def park(self):
        if self.num_wheels == 4:
            return 'In between the white lines!'
        else:
            return 'Oof, you better find a new spot :('
    def paint(self):
        return 'Added new ' + self.color + ' paint'
    def refill_gas_tank(self):
         self.gas = 10

jeep = Car('black', 'jeep')
```

Important Definitions

Instance attribute = A variable that is specific to a particular instance of a class

- In the car class, `color` and `make` are examples of instance attributes because they are set using self (`self.color, self.make`), so they are not the same for all cars

Class attribute = A variable that is shared across all instances of a class

- `num_wheels` is a class attribute because it is defined outside of the `__init__` method, and thus shared across all instances

Whenever we try to find the value of a particular attribute (such as `jeep.color`), we always check first in the instance attributes, then in the class attributes/methods, then in any parent class's class attributes/methods (see the inheritance section below).

Methods = Functions within a class

- `init`, `drive`, `park`, and `paint` are methods of the Car class

Dot Notation

When we access methods and variables in object oriented programming, we use what's called **dot notation**. Since we are now working with methods, not regular functions, we can no longer just say things like drive() and park() because we may be working with multiple classes that have a drive or a park method (and then which one would we choose??? who knows???). To access any method inside of a class, you can always do the following:

```
class_name.method(parameters)
```

For example, we can do `Car.drive(jeep)`, which will call `Car`'s `drive` method with `jeep` passed in as self. We can also access class attributes using similar notation:

```
class_name.class_attribute
```

An example of this would be `Car.num_wheels`, which would get the value of the `num_wheels` attribute of the `Car` class.

Now you may be wondering, what is self? Well, as a python convention, we generally use self within the methods of a class to refer to an instance of that particular class. That way, `self.some_attribute` signifies that we want the instance/class attribute (if there's no instance attribute) `some_attribute` associated with whatever we've passed in as self.

However, self is not a special keyword in python, so you could actually set the name of the first parameter of a method to be something that's not self (like foo, for example) and as long as you used `foo.some_attribute` instead of `self.some_attribute` throughout the method, your code would still work.

Technically, it is also possible to pass something that is not an instance of the current class in as self (such as the number 1), but depending on the body of the method, you may run into some errors if what you've passed in for self doesn't have all of the attributes that are requested in the method. For example, what if we tried to get `self.color` and we had passed in 1 for self? We would get an error.

Since self is the variable we use to represent an instance of the class, instead of doing `class_name.method(parameters)`, we can also call a method with `instance_name.method(any params other than self)`, which will first check the instance for any attributes called "method," then if it doesn't find anything, it will check the class that the instance is a part of to see if there are any methods called "method." We can do this because an instance of a class has access to all of the methods of its class.

For example, if we wanted to access jeep's color attribute, we would say:

```
jeep.color
```

And if we wanted to call the `drive` method of `Car` with `jeep` as self, we could say:

```
jeep.drive()
```

Notice that the parentheses after drive are empty because the drive method has no other parameters other than self. Also, note that if `jeep` was not an instance of `Car`, and was instead an instance of another class that had its own `drive` method, or if `jeep` had an instance attribute that was a function called `drive`, `Car.drive(jeep)` would not be the same as `jeep.drive()` because `jeep`'s `drive` would not be `Car`'s `drive` method.

Python's Magic Methods

It seems weird that every time we create an instance of a class, like when we defined jeep above, we just "called" the Car class, and it just knew to call the `__init__` method. Also, what's up with the double underscores?

Turns out that python has a bunch of "magic methods", written as __method_name__, that we've actually been working with since the very beginning of 61A. For each of these magic methods, there are shortcut ways to implicitly call them without writing out the full name with double underscores and everything. One example that we've used as a lot is `__eq__`, which is implicitly called every time we use the == operator.

In the case of `__init__`, we can implicitly call it by following a class name with parentheses (and passing in any non-self parameters as needed).

A special note about __init__ is that you never explicitly pass in anything for self. Self in the __init__ method always refers to the new instance that's being created, so you should pass in values for all parameters in __init__ except for self.

There are two other magic methods that are important for 61A: `__repr__` and `__str__`. The `__repr__` method of a class is implicitly called whenever we request an instance of that class. The `__str__` method of a class is implicitly called when we print an instance of that class. As with any of these methods, we can call them implicitly or explicitly, but `__repr__` and `__str__` perform slightly differently depending on whether the call is implicit or explicit. Let's look at an example to see what I mean:

```
class Car:
    num_wheels = 4
    def __init__(self, color):
        self.color = color
    def __str__(self):
        return "Love that " + self.color + " car!"
    def __repr__(self):
        return "Car has " + self.num_wheels + " wheels"
jeep = Car("powder blue")

>>> jeep
Car has 4 wheels
>>> jeep.__repr__()
"Car has 4 wheels"
>>> print(jeep)
Love that powder blue car!
>>> jeep.__str__()
"Love that powder blue car!"
```

Since jeep is an instance of car, requesting the variable jeep will implicitly call the `__repr__` method of the car class. Note that when we explicitly call `__repr__` , there are quotes, but when we implicitly call it there are no quotes. This is because when we implicitly call `__repr__`, we are actually printing the result of calling `__repr__` , and printing a string gets rid of the quotes. When we directly call it, there is no printing involved, so the quotes are still there.

Similarly, printing the variable jeep will implicitly call the `__str__` method of the Car class. When we explicitly call `__str__` , there are quotes, but when we implicitly call it there are no quotes. This is because when we call `__str__` implicitly, just as with `__repr__` , we are actually printing the result of calling `__str__` . When we directly call it, there is no printing involved, so the quotes are still there.

Inheritance

Sometimes, we want to create an object that shares some attributes/methods with another existing object. Rather than creating a whole new object and copying a lot of code from our existing object, we can make our new object inherit from our existing object. This is very similar to how we worked with higher order/parent functions earlier in the class, because a child class basically has access to all of the attributes and methods of its parent class. To

declare a child class, all you have to do is put the parent class name in parentheses after the child class name in the class definition.

For example, if we wanted to make the Motorcycle class inherit from the Car class:

```
class Motorcycle(Car):
    …
```

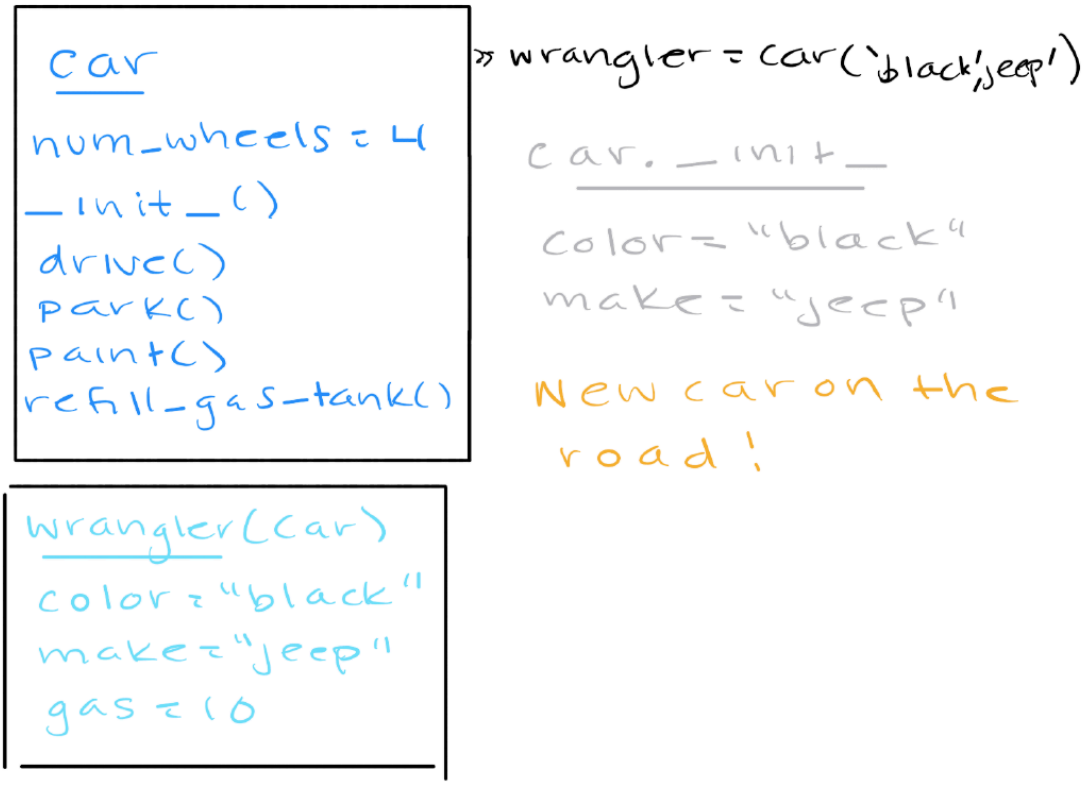
Now, any instance of the `Motorcycle` class has access to all of the attributes and methods of the `Car` class, so if we create an instance of the `Motorcycle` class called `yamaha`, we can do things like `yamaha.drive()`. We can also choose to override some of the attributes and methods from the `Car` class by redefining them in the `Motorcycle` class. If we decide that we want Motorcycles to only have two wheels instead of four, we can define a `num_wheels` attribute in the Motorcycle class and set it equal to 2. If we do that `yamaha.num_wheels` will return 2, but `jeep.num_wheels` will still return 4.

OOP Example Walkthrough

Because OOP can get a little complicated, I recommend creating a diagram to keep track of everything. I've made a walkthrough of how to best diagram and work through an OOP WWPD (which often shows up on midterm 2).

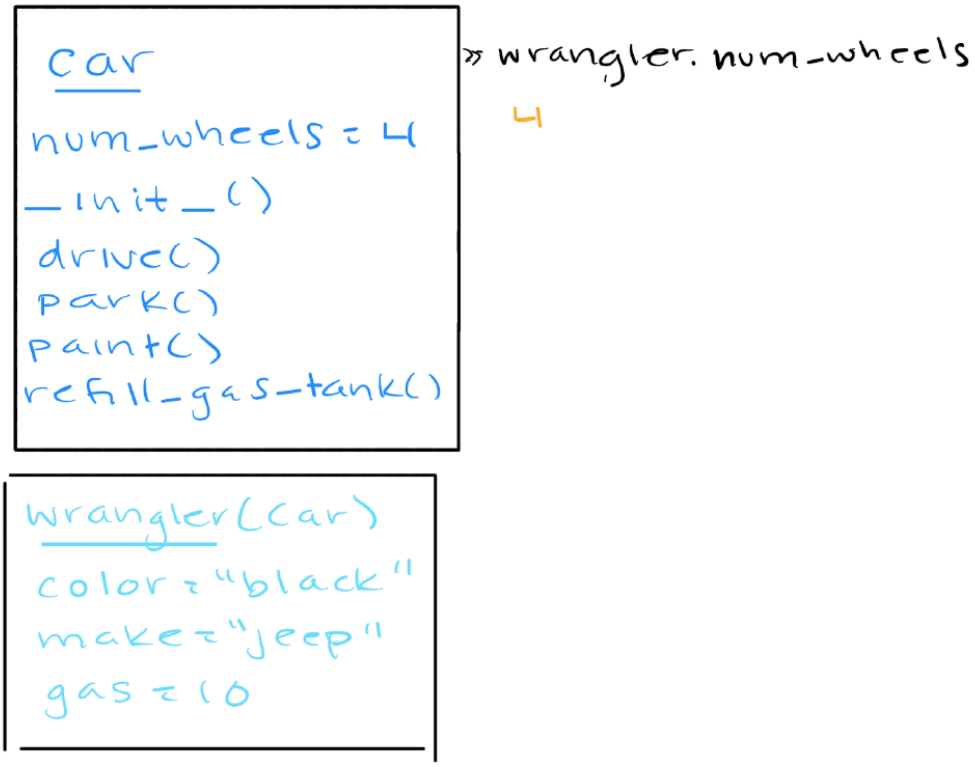
```
class Car:
    num_wheels = 4
    def __init__(self, color, make):
        self.color = color
        self.make = make
        self.gas = 10
        print("New car on the road!")
    def drive(self):
        if self.gas == 0:
            return "Can't drive on an empty tank!"
        self.gas -= 5
        return 'vroom vroom'
    def park(self):
        if num_wheels >= 4:
            return "In between the white lines!"
        else:
            return "Oof, you better find a new spot!"
    def paint(self):
        return "Added new " + self.color + " paint
            to the " + make + "!"
    def refill_gas_tank(self):
        self.gas = 10
class Motorcycle(Car):
    num_wheels = 2
```

```
>>> wrangler = Car('black', 'jeep')

>>> wrangler.num_wheels

>>> wrangler.drive(wrangler)

>>> Car.drive(wrangler)

>>> wrangler.num_wheels = 3
>>> acura = Car('silver', 'acura')

>>> lexus = Car('white', 'lexus')

>>> lexus.num_wheels

>>> Car.num_wheels = 5
>>> wrangler.num_wheels

>>> lexus.num_wheels

>>> acura.drive = lambda: 'Needs more gas!'
>>> acura.drive()

>>> Car.drive(acura)

>>> Car('neon green', 'toy').paint()

>>> harley = Motorcycle('orange', 'harley')

>>> harley.park()
```

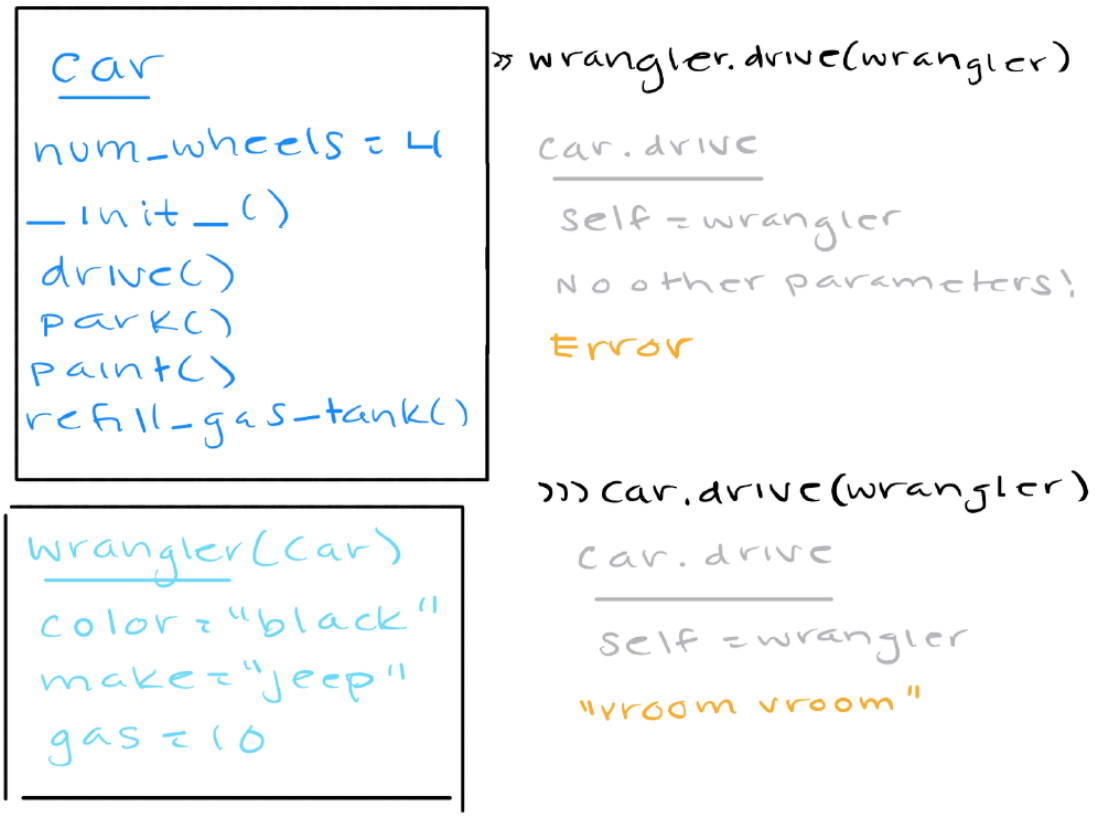


The first step of any OOP WWPD is to diagram any classes that you need to use. I like to diagram classes by creating a box for the class, then writing any class attributes/methods in the box. That way, you know all of the information that the class contains. For the above example, I wrote out a box representing the Car class, which contains its num_wheels attribute, as well at the __init__, drive, park, paint, and refill_gas_tank methods.

Now, looking at the WWPD code, we are setting wrangler equal to a new instance of the Car class, which we know is going to call the __init__ method of the car class. Car's __init__ takes in self (remember that we never explicitly pass in anything to __init__'s self parameter), color, which we've passed in "black" for, and make, which we've passed in "jeep" for. The __init__ method sets the color, make, and gas attributes, so I've created a new box representing wrangler, with all of those instance attribute values inside of it. We then print out "New car on the road!"

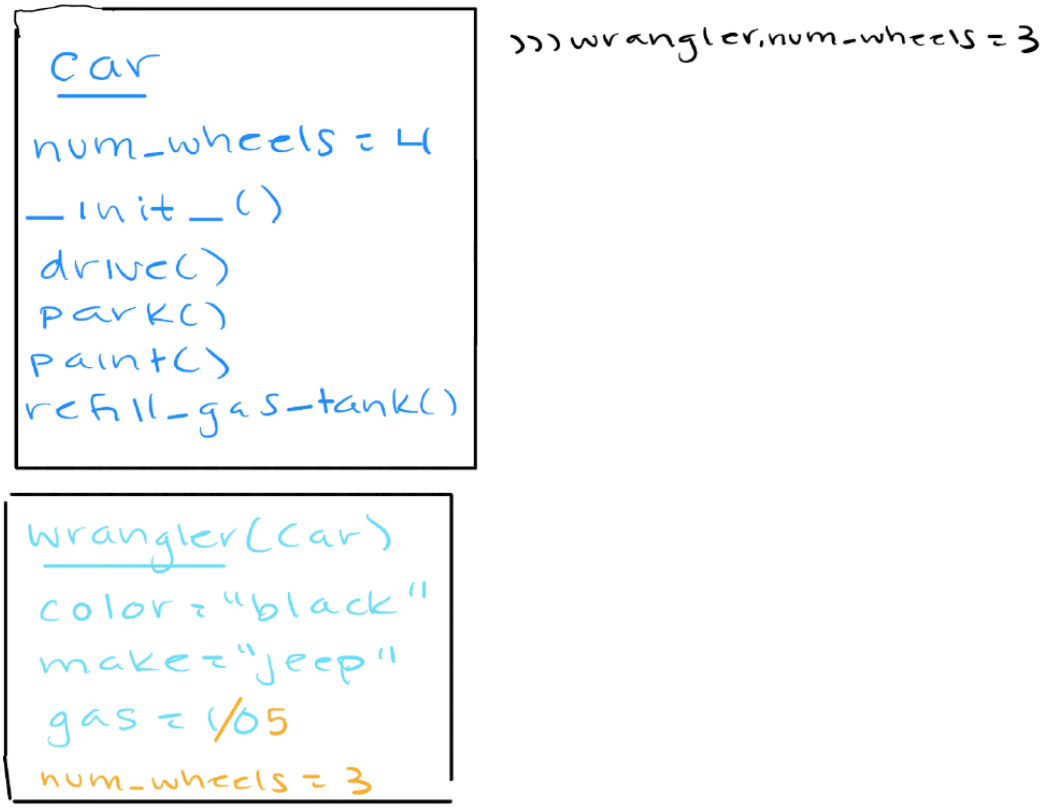


We will now start to see the value of diagramming: when we look for the value of wrangler.num_wheels, we can look in wrangler's box for a num_wheels. We don't see one, so we know it's not an instance attribute. We can then check Car's box, since wrangler is an instance of Car, and see that there is a num_wheels. We will return its value of 4.

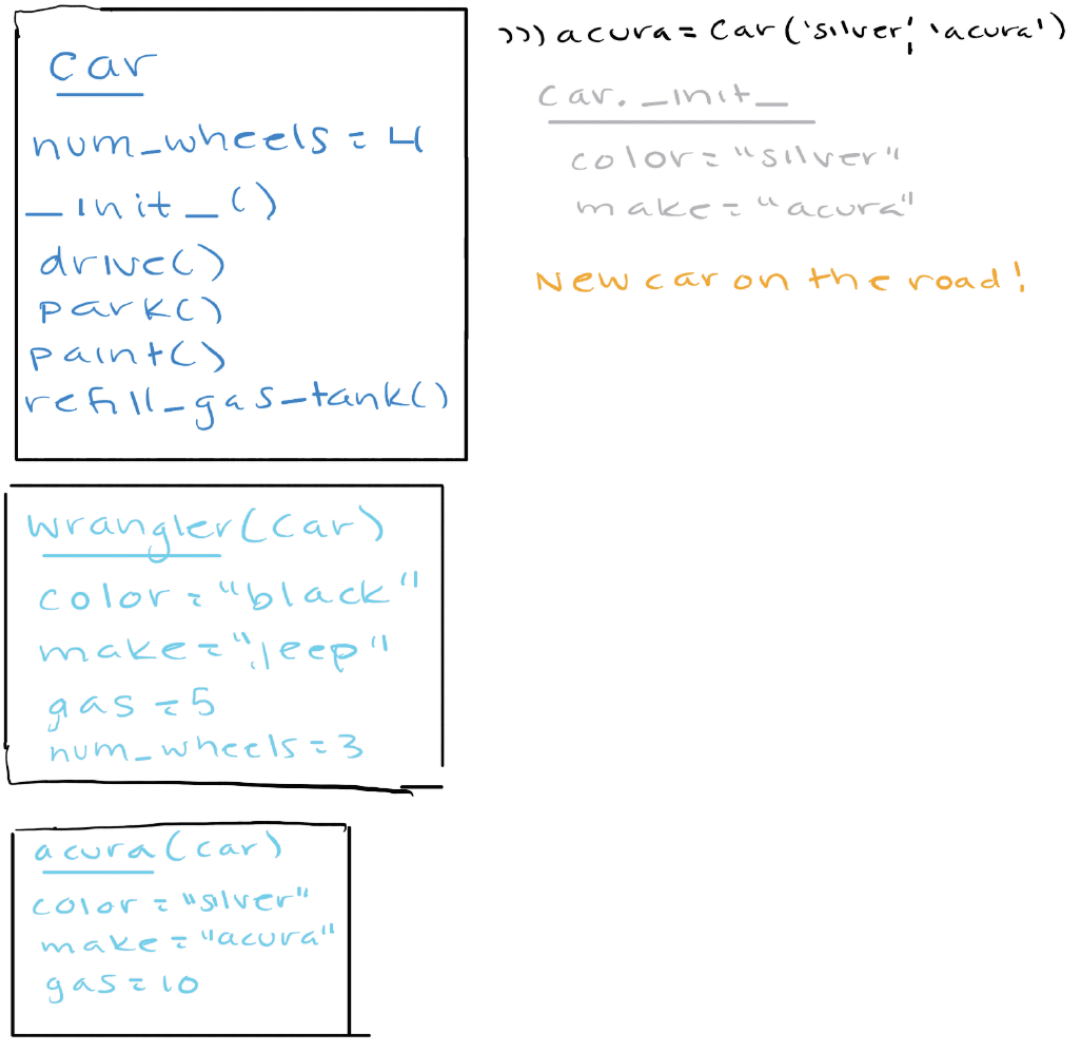


Now let's call some methods. Remember that wrangler.drive() means that we're calling the drive method of wrangler's class (which in this case means Car's drive method) with wrangler as self. But hang on, we've also passed in something in the parentheses as well. This will produce an error because we're trying to pass in two arguments (wrangler and wrangler), when there is only one parameter.

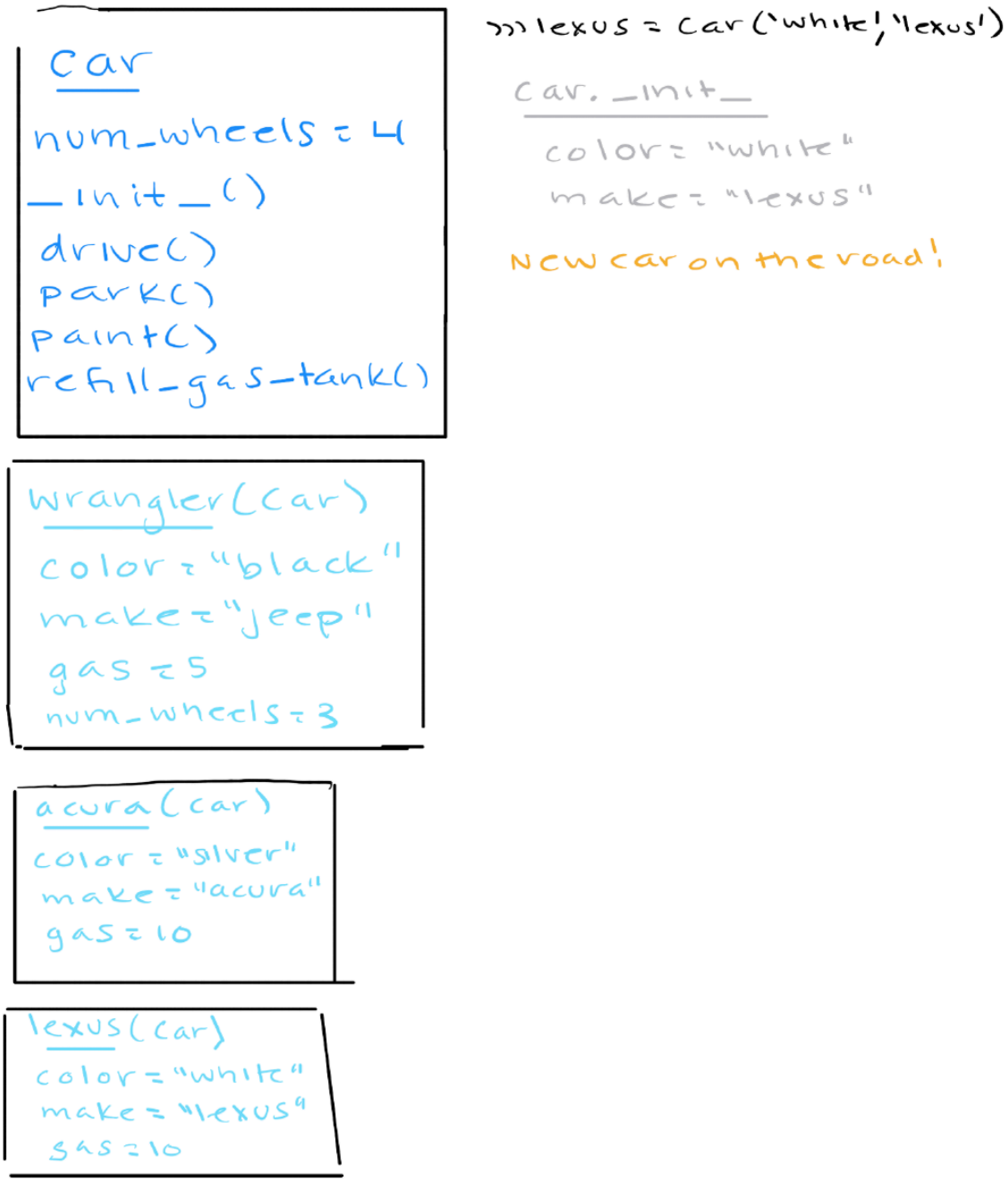
Car.drive(wrangler) does work, however, because we're calling Car.drive with wrangler passed in as self. That will first check if wrangler's gas = 0 (which it's not), then subtract 5 from wrangler's gas and return "vroom vroom." Note that we keep the quotes on "vroom vroom" because returning a string keeps the quotes on it.



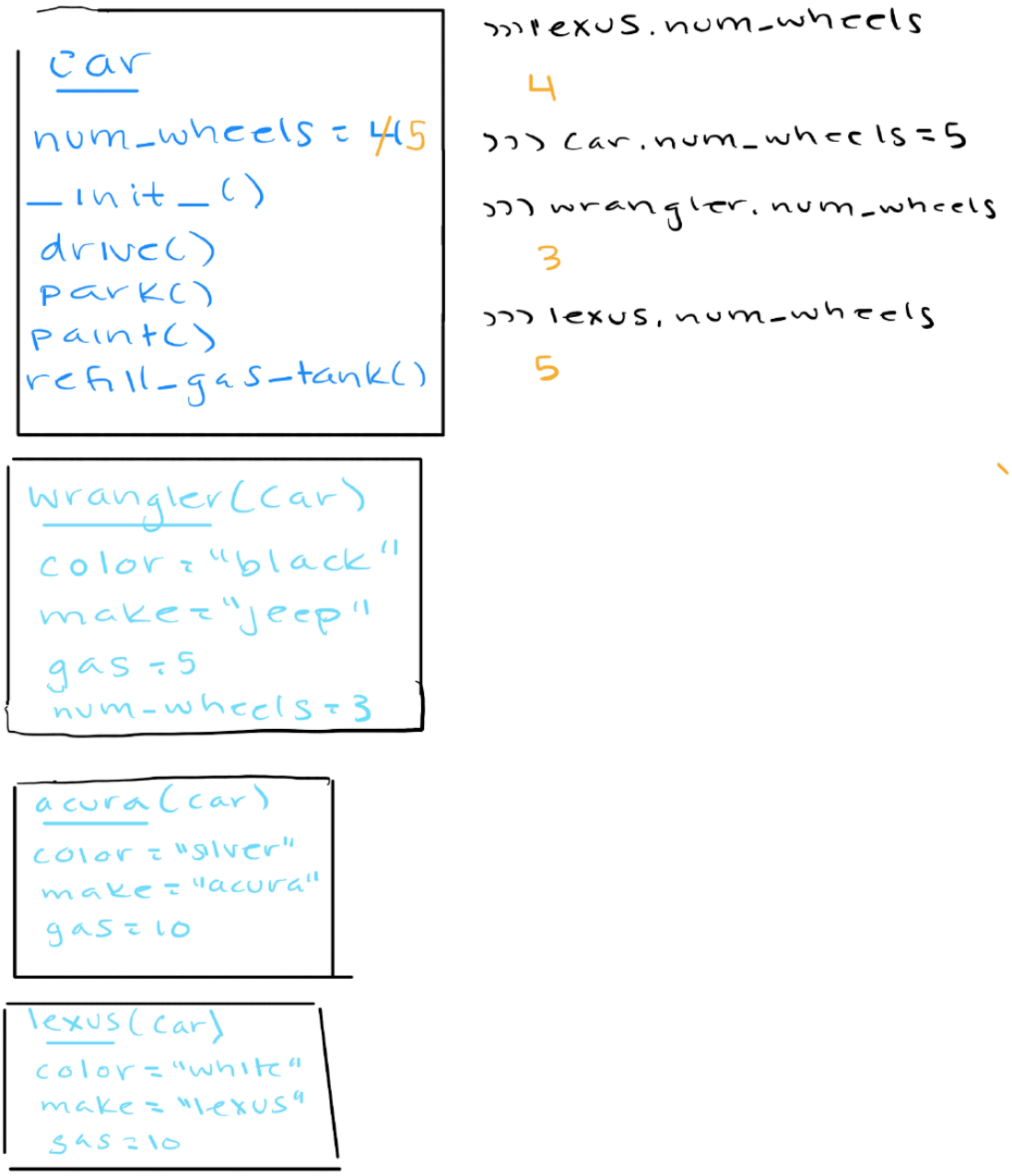
Now we need to set a num_wheels attribute in wrangler. Note that we are not changing the value of Car.num_wheels! We explicitly state that we want to set wrangler.num_wheels, so we will create a new instance attribute in wrangler called num_wheels and set it equal to 3.



We are now creating a new instance of the Car class, which means that we call Car's __init__ method. Car's __init__ takes in self (remember that we never explicitly pass in anything to __init__'s self parameter), color, which we've passed in "silver" for, and make, which we've passed in "acura" for. The __init__ method sets the color, make, and gas attributes, so I've created a new box representing acura, with all of those instance attribute values inside of it. We then print out "New car on the road!"

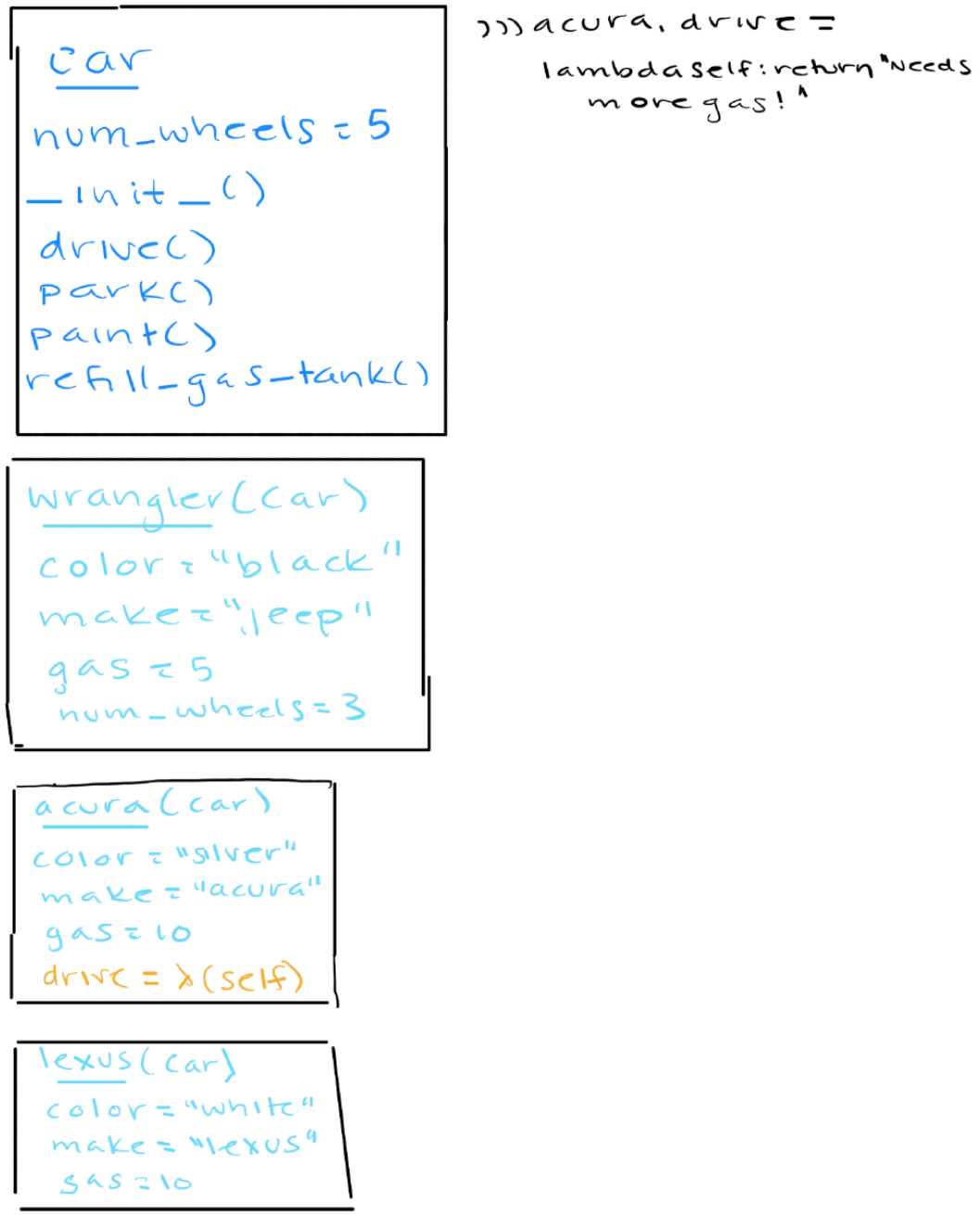


We are now creating another new instance of the Car class, which means that we call Car's __init__ method. Car's __init__ takes in self (remember that we never explicitly pass in anything to __init__'s self parameter), color, which we've passed in "white" for, and make, which we've passed in "lexus" for. The __init__ method sets the color, make, and gas attributes, so I've created a new box representing lexus, with all of those instance attribute values inside of it. We then print out "New car on the road!"



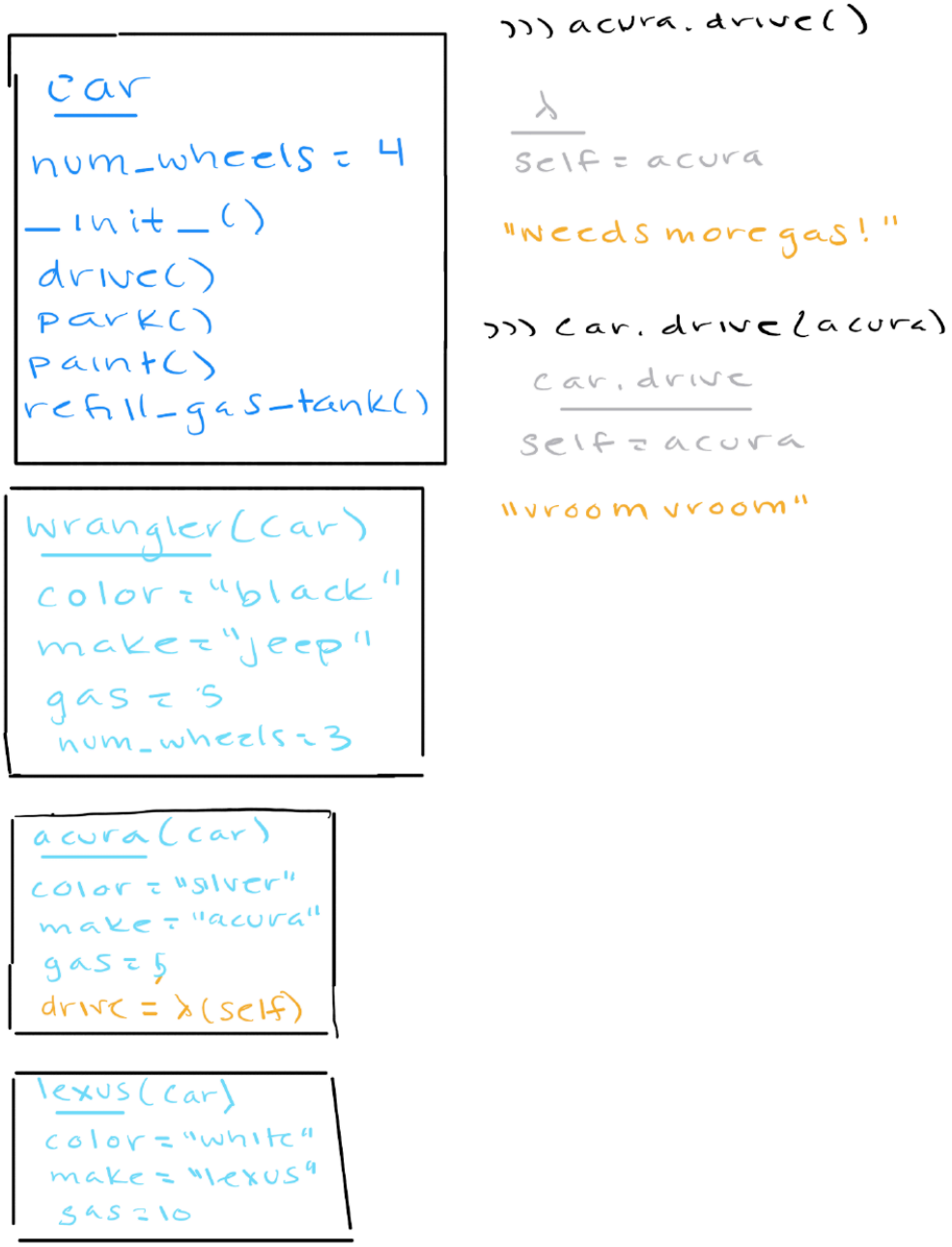
When we look for the value of lexus.num_wheels, we can look in lexus's box for a num_wheels. We don't see one, so we know it's not an instance attribute. We can then check Car's box, since lexus is an instance of Car, and see that there is a num_wheels. We will return its value of 4.

Next, we update Car's num_wheels attribute to 5. Note that when we call wrangler.num_wheels, we still get 3, because we check instance attributes before class attributes, and wrangler has an instance attribute num_wheels with a value of 3. However, when we call lexus.num_wheels, we can see that it has changed to 5, because we're pulling its value from Car.num_wheels, which is now 5.

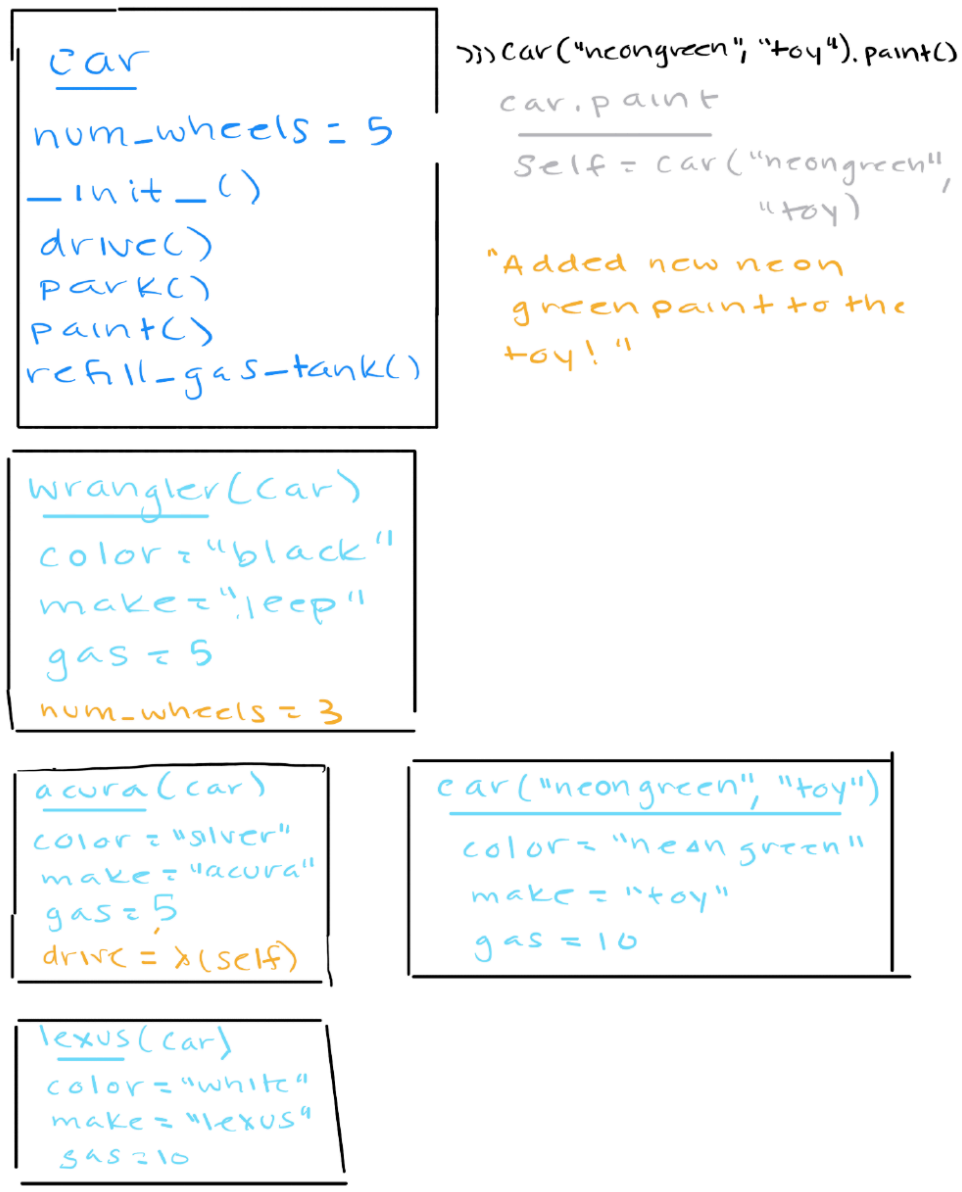


Just as when we set wrangler.num_wheels, we will also create a new instance attribute for acura called drive, set that equal to the given lambda function, and add it to acura's box.

Note: I actually made a typo when writing out the code for this, look at the above code to see the correct line of code. It should be `acura.drive = lambda: "Needs more gas!"` Ignore the lambda parameter.

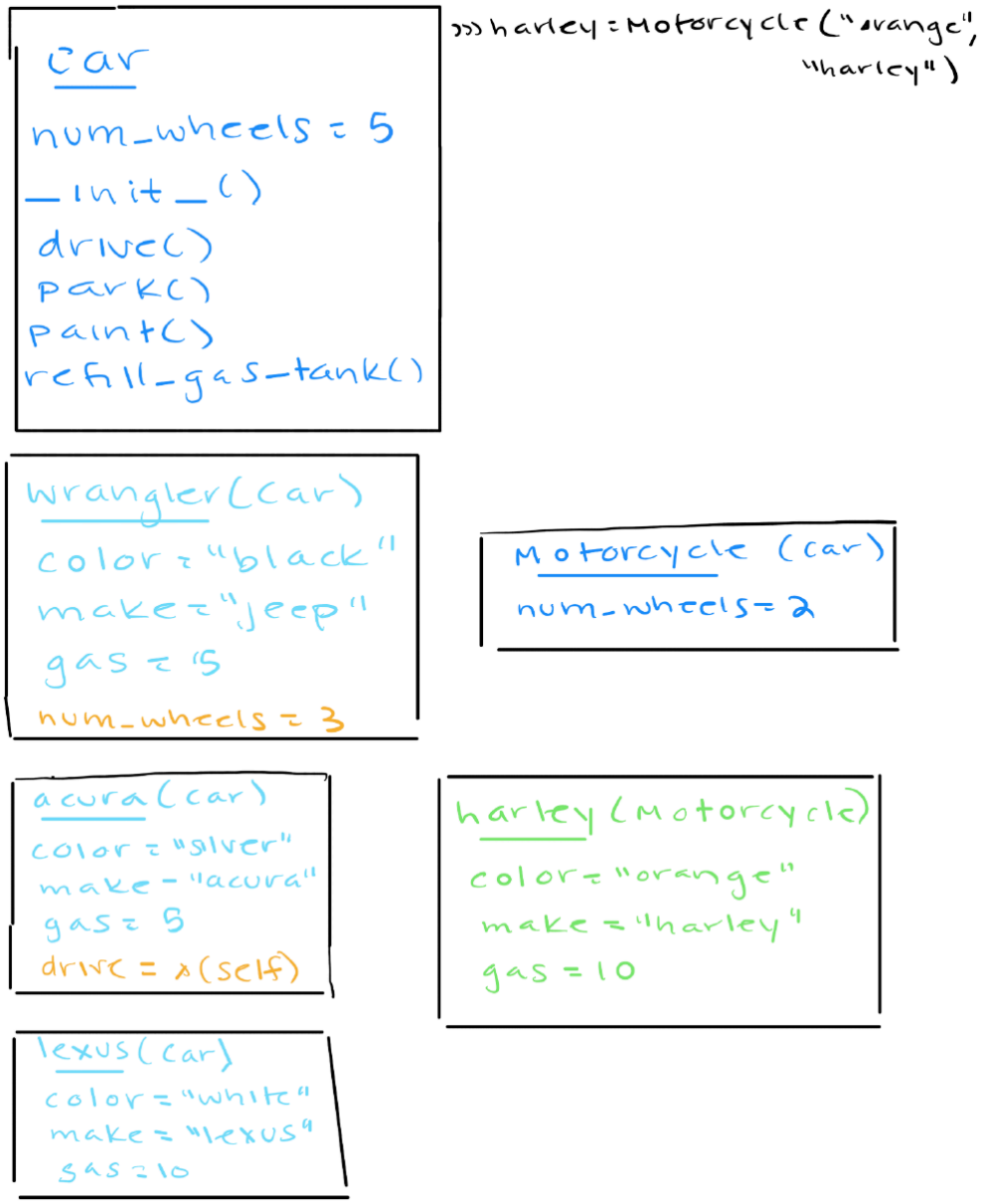


Now is where we really see the subtle difference between acura.drive() and Car.drive(acura): when we search for acura's drive, we will first look inside the instance, and find the lambda function. We will then run that as drive, returning "Needs more gas!" When we call Car.drive(acura), we will actually run the body of Car.drive, which will subtract 5 from acura's gas and return "vroom vroom."

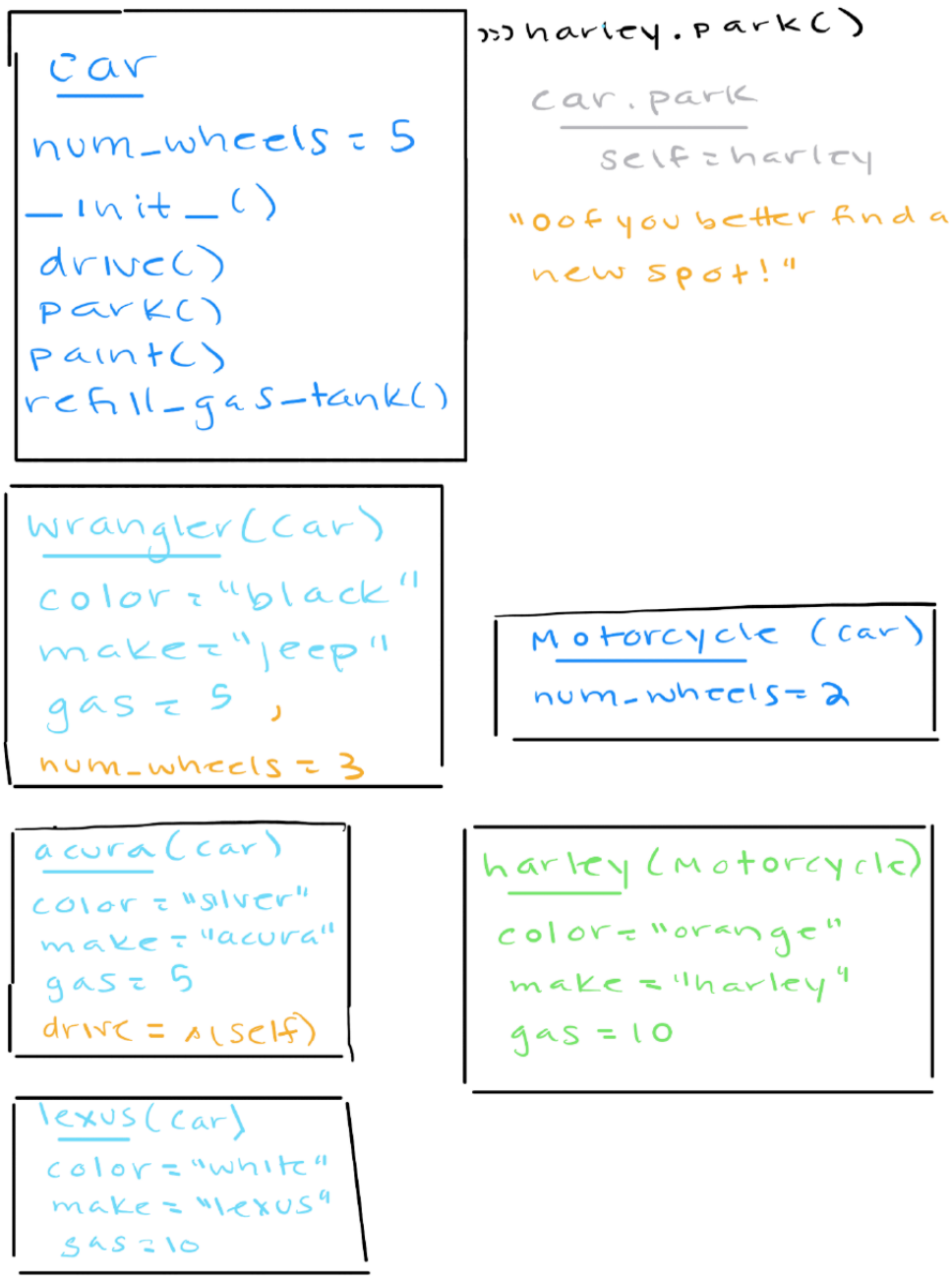


We are now creating another new instance of the Car class, which means that we call Car's __init__ method. Note, however, that we didn't assign this instance to any variable name, so once we create the instance and run the paint method, the instance will go away because it's not bound to any variable name. Car's __init__ takes in self (remember that we never explicitly pass in anything to __init__'s self parameter), color, which we've passed in "neon green" for, and make, which we've passed in "toy" for. The __init__ method sets the color, make, and gas attributes, so I've created a new box representing this unnamed instance, with all of those instance attribute values inside of it. We then print out "New car on the road!"

Now, to run the paint method, we first check the instance, find no paint attribute, then check the Car class and find the paint method. It returns the string "Added new neon green paint to the toy!"



Since we're creating our first instance of the Motorcycle class, I also wrote out a box representing the Motorcycle class. The only thing it has inside of it is the num_wheels attribute, which is equal to 2. Note also that the Motorcycle class inherits from the Car class, so when we create harley, we will check the Motorcycle class for an __init__, and since it doesn't have one, we will check the Car class, which does have one. We will set the instance attributes color, make, and gas for Harley (which I've added to a new box), and print "New car on the road!"



When we call harley.park(), we will first check harley for any instance attributes called park, then check the Motorcycle class for a park, then check the Car class, which does have one. Self will be harley, so when we look for the value of harley.num_wheels, we will first check the instance (which doesn't have a num_wheels), then check the Motorcycle class, which does have one. Since Motorcycle.num_wheels = 2, we will return "Oof you better find a new spot!"