1 Objects

Object Oriented Programming

Object-Oriented Programming (OOP) allows us to treat data as objects like we do in real life.

For example, consider the class Student. Each of you as individuals are an instance of this class. So, a student Angela would be an instance of the class Student. Details that all CS 61A students have, such as name, year, and major, are called instance attributes. Every student has these attributes, but their values differ from student to student. An attribute that is shared among all instances of Student is known as a class attribute. An example would be the instructors attribute; the instructors for CS 61A, DeNero and Hilfinger, are the same for every student in CS 61A.

All students are able to do homework, attend lecture, and go to office hours. When functions belong to a specific object, they are said to be methods. In this case, these actions would be bound methods of Student objects.

Here is a recap of what we discussed above:

- class: a template for creating objects
- instance: a single object created from a class
- instance attribute: a property of an object, specific to an instance
- class attribute: a property of an object, shared by all instances of a class
- method: an action (function) that all instances of a class may perform

OOP and Inheritance Example: Cars

Professor Hilfinger is running late, and needs to get from San Francisco to Berkeley before lecture starts. He'd take BART, but that will take too long. It'd be great if he had a car. A monster truck would be best, but a car will do – for now...

In car.py, you'll find a class called Car. A class is a blueprint for creating objects of that type. In this case, the Car class statement tells us how to create Car objects.

So let's build Professor Hilfinger a car! Don't worry, you won't need to do any physical work – the constructor will do it for you. The constructor of a class is a function that creates an instance, or one single occurrence, of the object outlined by the class. In Python, the constructor method is named __init__. Note that there must be two underscores on each side of init. The Car class' constructor looks like this:

```
def __init__(self, make, model):
    self.make = make
    self.model = model
    self.color = 'No color yet. You need to paint me.'
    self.wheels = Car.num_wheels
    self.gas = Car.gas
```

The __init__ method for Car has three parameters. The first one, self, is automatically bound to the newly created Car object. The second and third parameters, make and model, are bound to the arguments passed to the constructor, meaning when we make a Car object, we must provide two arguments. Don't worry about the code inside the body of the constructor for now.

Let's make our car. Professor Hilfinger would like to drive a Tesla Model S to lecture. We can construct an instance of Car with 'Tesla' as the make and 'Model S' as the model. Rather than calling __init__ explicitly, Python allows us to make an instance of a class by using the name of the class.

```
>>> hilfingers_car = Car('Tesla', 'Model S')
```

Here, 'Tesla' is passed in as the make, and 'Model S' as the model. Note that we don't pass in an argument for self, since its value is always the object being created. An object is an instance of a class. In this case, hilfingers_car is now bound to a Car object or, in other words, an instance of the Car class.

So how are the make and model of Professor Hilfinger's car actually stored? Let's talk about attributes of instances and classes. Here's a snippet of the code in car.py of the instance and class attributes in the Car class:

```
class Car(object):
    num_wheels = 4
    gas = 30
    headlights = 2
    size = 'Tiny'

    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.color = 'No color yet. You need to paint me.'
        self.wheels = Car.num_wheels
        self.gas = Car.gas

    def paint(self, color):
        self.color = color
        return self.make + ' ' + self.model + ' is now ' + color
```

In the first two lines of the constructor, the name self.make is bound to the first argument passed to the constructor and self.model is bound to the second. These are two examples of instance attributes. An instance attribute is a quality that is specific to an instance, and thus can only be accessed using dot notation (separating the instance and attribute with a period) on an instance. In this case, self is bound to our instance, so self.model references our instance's model.

Our car has other instance attributes too, like color and wheels. As instance attributes, the make, model, and color of hilfingers_car do not affect the make, model, and color of other cars.

On the other hand, a class attribute is a quality that is shared among all instances of the class. For example, the Car class has four class attributes defined at the beginning of a class: num_wheels = 4, gas = 30, headlights = 2 and size = 'Tiny'. The first says that all cars have 4 wheels.

You might notice in the __init__ method of the Car class, the instance attribute gas is initialized to the value of Car.gas, the class attribute. Why don't we just use the class attribute, then? The reason is because each Car's gas attribute needs to be able to change independently of each other. If one Car drives for a while, it should use up some gas, and that Car instance should reflect that by having a lower gas value. However, all other Cars shouldn't lose any gas, and changes to a class attribute will affect all instances of the class.

Class attributes can also be accessed using dot notation, both on an instance and on the class name itself. For example, we can access the class attribute size of Car like this:

```
>>> Car.size
'Tiny'
```

And in the following line, we access hilfingers_car's color attribute:

```
>>> hilfingers_car.color
'No color yet. You need to paint me.'
```

Looks like we need to paint hilfingers_car!

Let's use the paint method from the Car class. Methods are functions that are specific to a class; only an instance of the class can use them. We've already seen one method: __init__! Think of methods as actions or abilities of objects. How do we call methods on an instance? You guessed it, dot notation!

```
>>> hilfingers_car.paint('black')
'Tesla Model S is now black'
>>> hilfingers_car.color
'black'
```

Awesome! But if you take a look at the paint method, it takes two parameters. So why don't we need to pass two arguments? Just like we've seen with __init__, all methods of a class have a self parameter to which Python automatically binds the instance that is calling that method. Here, hilfingers_car is bound to self so that the body of paint can access its attributes!

You can also call methods using the class name and dot notation; for example,

```
>>> Car.paint(hilfingers_car, 'red')
'Tesla Model S is now red'
```

Notice that unlike when we painted Professor Hilfinger's car black, this time we had to pass in two arguments: one for self and one for color. This is because when you call a method using dot notation from an instance, Python knows what instance to automatically bind to self. However, when you call a method using dot notation from the class, Python doesn't know which instance of Car we want to paint, so we have to pass that in as well.

Professor Hilfinger's red Tesla is pretty cool, but it has to sit in traffic. How about we create a monster truck for him instead? In car.py, we've defined a MonsterTruck class. Let's look at the code for MonsterTruck:

```
class MonsterTruck(Car):
    size = 'Monster'

    def rev(self):
        print('Vroom! This Monster Truck is huge!')

    def drive(self):
        self.rev()
        return Car.drive(self)
```

Wow! The truck may be big, but the source code is tiny! Let's make sure that the truck still does what we expect it to do. Let's create a new instance of Professor Hilfinger's monster truck:

```
>>> hilfingers_truck = MonsterTruck('Monster Truck', 'XXL')
```

Does it behave as you would expect a Car to? Can you still paint it? Is it even drivable?

Well, the class MonsterTruck is defined as class MonsterTruck(Car):, meaning its superclass is Car. Likewise, the class MonsterTruck is a subclass of the Car class. That means the MonsterTruck class inherits all the attributes and methods that were defined in Car, including its constructor!

Inheritance makes setting up a hierarchy of classes easier because the amount of code you need to write to define a new class of objects is reduced. You only need to add (or override) new attributes or methods that you want to be unique from those in the superclass.

```
>>> hilfingers_car.size
'Tiny'
>>> hilfingers_truck.size
'Monster'
```

Wow, what a difference in size! This is because the class attribute size of MonsterTruck overrides the size class attribute of Car, so all MonsterTruck instances are 'Monster'-sized.

In addition, the drive method in MonsterTruck overrides the one in Car. To show off all MonsterTruck instances, we defined a rev method specific to MonsterClass. Regular Cars cannot rev! Everything else – the constructor __init__, paint, num_wheels, gas – are inherited from Car.

2 Nonlocal

Consider the following function:

```
def make_counter():
    """Makes a counter function.

    >>> counter = make_counter()
    >>> counter()
    1
    >>> counter()
    2
    """
    count = 0
    def counter():
        count = count + 1
        return count
    return counter
```

Running this function's doctests, we find that it causes the following error:

```
UnboundLocalError: local variable 'count' referenced before assignment
```

Why does this happen? When we execute an assignment statement, remember that we are either creating a new binding in our current frame or we are updating an old one in the current frame. For example, the line count = ... in counter, is creating the local variable count inside counter's frame. This assignment statement tells Python to expect a variable called count inside counter's frame, so Python will not look in parent frames for this variable. However, notice that we tried to compute count + 1 before the local variable was created! That's why we get the UnboundLocalError.

To avoid this problem, we introduce the nonlocal keyword. It allows us to update a variable in a parent frame!
Note: we cannot use nonlocal to modify variables in the global frame.

Consider this improved example:

```
 def make_counter():
    """Makes a counter function.

    >>> counter = make_counter()
    >>> counter()
    1
    >>> counter()
    2
    """
    count = 0
    def counter():
        nonlocal count
        count = count + 1
        return count
    return counter
```

The line nonlocal count tells Python that count will not be local to this frame, so it will look for it in parent frames. Now we can update count without running into problems.

Examples for Nonlocal

Question 1

```
>>> a = 0
>>> def f():
...     nonlocal a
...     a += 1
...
```

Question 2

```
>>> def f():
...     a = 0
...     def g():
...         a = 1
...         def h():
...             nonlocal a
...             return a
...         return h
...     return g
...
>>> f()()()
```

Question 3

```
>>> eggplant = 8
>>> def vegetable(kale):
...     def eggplant(spinach):
...         nonlocal eggplant
...         nonlocal kale
...         kale = 9
...         eggplant = spinach
...         print(eggplant, kale)
...     eggplant(kale)
...     return eggplant
...
>>>spinach = vegetable('kale')
```