Jeff Knupp

PYTHON PROGRAMMER

Everything I know about Python...

Learn to Write Pythonic Code!

Check out the book *Writing Idiomatic Python*!
(https://www.jeffknupp.com/writing-idiomatic-python-ebook/)

**Looking for Python Tutoring? Remote and local (NYC) slots still available!
Email me at jeff@jeffknupp.com (mailto:jeff@jeffknupp.com) for more info.**

Improve Your Python: Python Classes and Object Oriented Programming (/blog/2014/06/18/improve-your-python-python-classes-and-object-oriented-programming/)

The `class` is a fundamental building block in Python. It is the underpinning for not only many popular programs and libraries, but the Python standard library as well. Understanding what classes are, when to use them, and how they can be useful is essential, and the goal of this article. In the process, we'll explore what the term *Object-Oriented Programming* means and how it ties together with Python classes.

Everything Is An Object...

What is the `class` keyword used for, exactly? Like its function-based cousin `def`, it concerns the *definition* of things. While `def` is used to define a function, `class` is used to define a *class*. And what is a class? Simply a logical grouping of data and functions (the latter of which are frequently referred to as "methods" when defined within a class).

What do we mean by "logical grouping"? Well, a class can contain any data we'd like it to, and can have any functions (methods) attached to it that we please. Rather than just throwing random things together under the name "class", we try to create classes where there is a logical connection between things. Many times, classes are based on objects in the real world (like `Customer` or `Product`). Other times, classes are based on concepts in our system, like `HTTPRequest` or `Owner`.

Regardless, classes are a *modeling* technique; a way of thinking about programs. When you think about and implement your system in this way, you're said to be performing *Object-Oriented Programming*. "Classes" and "objects" are words that are often used interchangeably, but they're not really the same thing. Understanding what makes them different is the key to understanding what they are and how they work.

..So Everything Has A Class?

Classes can be thought of as *blueprints for creating objects*. When I *define* a Customer class using the `class` keyword, I haven't actually created a customer. Instead, what I've created is a sort of instruction manual for constructing "customer" objects. Let's look at the following example code:

```
class Customer(object):
    """A customer of ABC Bank with a checking account. Customers have the
    following properties:

    Attributes:
        name: A string representing the customer's name.
        balance: A float tracking the current balance of the customer's account.
    """

    def __init__(self, name, balance=0.0):
        """Return a Customer object whose name is *name* and starting
        balance is *balance*."""
        self.name = name
        self.balance = balance

    def withdraw(self, amount):
        """Return the balance remaining after withdrawing *amount*
        dollars."""
        if amount > self.balance:
            raise RuntimeError('Amount greater than available balance.')
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        """Return the balance remaining after depositing *amount*
        dollars."""
        self.balance += amount
        return self.balance
```

The `class Customer(object)` line *does not* create a new customer. That is, just because we've *defined* a `Customer` doesn't mean we've *created* one; we've merely outlined the *blueprint* to create a `Customer` object. To do so, we call the class's `__init__` method with the proper number of arguments (minus `self`, which we'll get to in a moment).

So, to use the "blueprint" that we created by defining the `class Customer` (which is used to create `Customer` objects), we call the class name almost as if it were a function: `jeff = Customer('Jeff Knupp', 1000.0)`. This line simply says "use the `Customer` blueprint to create me a new object, which I'll refer to as `jeff`."

The `jeff` *object*, known as an *instance*, is the realized version of the `Customer` *class*. Before we called `Customer()`, no `Customer` object existed. We can, of course, create as many `Customer` objects as we'd like. There is still, however, only one `Customer` *class*, regardless of how many *instances* of the class we create.

`self`?

So what's with that `self` parameter to all of the `Customer` methods? What is it? Why, it's the instance, of course! Put another way, a method like `withdraw` defines the instructions for withdrawing money from *some abstract customer's account*. Calling `jeff.withdraw(100.0)` puts those instructions to use *on the `jeff` instance*.

So when we say `def withdraw(self, amount):`, we're saying, "here's how you withdraw money from a Customer object (which we'll call `self`) and a dollar figure (which we'll call `amount`). `self` is the *instance* of the `Customer` that `withdraw` is being called on. That's not me making analogies, either. `jeff.withdraw(100.0)` is just shorthand for `Customer.withdraw(jeff, 100.0)`, which is perfectly valid (if not often seen) code.

`__init__`

`self` may make sense for other methods, but what about `__init__`? When we call `__init__`, we're in the process of creating an object, so how can there already be a `self`? Python allows us to extend the `self` pattern to when objects are constructed as well, even though it doesn't *exactly* fit. Just imagine that `jeff = Customer('Jeff Knupp', 1000.0)` is the same as calling `jeff = Customer(jeff, 'Jeff Knupp', 1000.0)`; the `jeff` that's passed in is also made the result.

This is why when we call `__init__`, we *initialize* objects by saying things like `self.name = name`. Remember, since `self` *is* the instance, this is equivalent to saying `jeff.name = name`, which is the same as `jeff.name = 'Jeff Knupp'`. Similarly, `self.balance = balance` is the same as `jeff.balance = 1000.0`. After these two lines, we consider the `Customer` object "initialized" and ready for use.

Be careful what you `__init__`

After `__init__` has finished, the caller can rightly assume that the object is ready to use. That is, after `jeff = Customer('Jeff Knupp', 1000.0)`, we can start making `deposit` and `withdraw` calls on `jeff`; `jeff` is a **fully-initialized** object.

Imagine for a moment we had defined the `Customer` class slightly differently:

```
class Customer(object):
    """A customer of ABC Bank with a checking account. Customers have the
    following properties:

    Attributes:
        name: A string representing the customer's name.
        balance: A float tracking the current balance of the customer's account.
    """

    def __init__(self, name):
        """Return a Customer object whose name is *name*."""
        self.name = name

    def set_balance(self, balance=0.0):
        """Set the customer's starting balance."""
        self.balance = balance

    def withdraw(self, amount):
        """Return the balance remaining after withdrawing *amount*
        dollars."""
        if amount > self.balance:
            raise RuntimeError('Amount greater than available balance.')
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        """Return the balance remaining after depositing *amount*
        dollars."""
        self.balance += amount
        return self.balance
```

This may look like a reasonable alternative; we simply need to call `set_balance` before we begin using the instance. There's no way, however, to communicate this to the caller. Even if we document it extensively, we can't *force* the caller to call `jeff.set_balance(1000.0)` before calling `jeff.withdraw(100.0)`. Since the `jeff` instance doesn't even *have* a balance attribute until `jeff.set_balance` is called, this means that the object hasn't been "fully" initialized.

The rule of thumb is, don't *introduce* a new attribute outside of the `__init__` method, otherwise you've given the caller an object that isn't fully initialized. There are exceptions, of course, but it's a good principle to keep in mind. This is part of a larger concept of object *consistency*: there shouldn't be any series of method calls that can result in the object entering a state that doesn't make sense.

Apply For a I
Ad Apply For E
Master's Degre
Rome Business Scho
Open

Invariants (like, "balance should always be a non-negative number") should hold both when a method is entered and when it is exited. It should be impossible for an object to get into an invalid state just by calling its methods. It goes without saying, then, that an object should *start* in a valid state as well, which is why it's important to initialize everything in the `__init__` method.

Instance Attributes and Methods

An function defined in a class is called a "method". Methods have access to all the data contained on the instance of the object; they can access and modify anything previously set on `self`. Because they use `self`, they require an instance of the class in order to be used. For this reason, they're often referred to as "instance methods".

If there are "instance methods", then surely there are other types of methods as well, right? Yes, there are, but these methods are a bit more esoteric. We'll cover them briefly here, but feel free to research these topics in more depth.

Static Methods

Class attributes are attributes that are set at the *class-level*, as opposed to the *instance-level*. Normal attributes are introduced in the `__init__` method, but some attributes of a class hold for *all* instances in all cases. For example, consider the following definition of a `Car` object:

```
1  class Car(object):
2
3      wheels = 4
4
5      def __init__(self, make, model):
6          self.make = make
7          self.model = model
8
9  mustang = Car('Ford', 'Mustang')
10 print mustang.wheels
11 # 4
12 print Car.wheels
13 # 4
```

A `Car` always has four `wheels`, regardless of the `make` or `model`. Instance methods can access these attributes in the same way they access regular attributes: through `self` (i.e. `self.wheels`).

There is a class of methods, though, called *static methods*, that don't have access to `self`. Just like class attributes, they are methods that work without requiring an instance to be present. Since instances are always referenced through `self`, static methods have no `self` parameter.

The following would be a valid static method on the `Car` class:

```
class Car(object):
    ...
    def make_car_sound():
        print 'VRoooooommmm!'
```

No matter what kind of car we have, it always makes the same sound (or so I tell my ten month old daughter). To make it clear that this method should not receive the instance as the first parameter (i.e. `self` on "normal" methods), the `@staticmethod` decorator is used, turning our definition into:

```
class Car(object):
    ...
    @staticmethod
    def make_car_sound():
        print 'VRoooooommmm!'
```

Class Methods

A variant of the static method is the *class method*. Instead of receiving the *instance* as the first parameter, it is passed the *class*. It, too, is defined using a decorator:

```
class Vehicle(object):
    ...
    @classmethod
    def is_motorcycle(cls):
        return cls.wheels == 2
```

Class methods may not make much sense right now, but that's because they're used most often in connection with our next topic: *inheritance*.

Inheritance

While Object-oriented Programming is useful as a modeling tool, it truly gains power when the concept of *inheritance* is introduced. *Inhertitance* is the process by which a "child" class *derives* the data and behavior of a "parent" class. An example will definitely help us here.

Imagine we run a car dealership. We sell all types of vehicles, from motorcycles to trucks. We set ourselves apart from the competition by our prices. Specifically, how we determine the price of a vehicle on our lot: $5,000 x number of wheels a vehicle has. We love buying back our vehicles as well. We offer a flat rate - 10% of the miles driven on the vehicle. For trucks, that rate is $10,000. For cars, $8,000. For motorcycles, $4,000.

If we wanted to create a sales system for our dealership using Object-oriented techniques, how would we do so? What would the objects be? We might have a `Sale` class, a `Customer` class, an `Inventory` class, and so forth, but we'd almost certainly have a `Car`, `Truck`, and `Motorcycle` class.

What would these classes look like? Using what we've learned, here's a possible implementation of the `Car` class:

```
class Car(object):
    """A car for sale by Jeffco Car Dealership.

    Attributes:
        wheels: An integer representing the number of wheels the car has.
        miles: The integral number of miles driven on the car.
        make: The make of the car as a string.
        model: The model of the car as a string.
        year: The integral year the car was built.
        sold_on: The date the vehicle was sold.
    """

    def __init__(self, wheels, miles, make, model, year, sold_on):
        """Return a new Car object."""
        self.wheels = wheels
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on

    def sale_price(self):
        """Return the sale price for this car as a float amount."""
        if self.sold_on is not None:
            return 0.0  # Already sold
        return 5000.0 * self.wheels

    def purchase_price(self):
        """Return the price for which we would pay to purchase the car."""
        if self.sold_on is None:
            return 0.0  # Not yet sold
        return 8000 - (.10 * self.miles)

    ...
```

OK, that looks pretty reasonable. Of course, we would likely have a number of other methods on the class, but I've shown two of particular interest to us: `sale_price` and `purchase_price`. We'll see why these are important in a bit.

Now that we've got the `Car` class, perhaps we should crate a `Truck` class? Let's follow the same pattern we did for car:

```
class Truck(object):
    """A truck for sale by Jeffco Car Dealership.

    Attributes:
        wheels: An integer representing the number of wheels the truck has.
        miles: The integral number of miles driven on the truck.
        make: The make of the truck as a string.
        model: The model of the truck as a string.
        year: The integral year the truck was built.
        sold_on: The date the vehicle was sold.
    """

    def __init__(self, wheels, miles, make, model, year, sold_on):
        """Return a new Truck object."""
        self.wheels = wheels
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on

    def sale_price(self):
        """Return the sale price for this truck as a float amount."""
        if self.sold_on is not None:
            return 0.0  # Already sold
        return 5000.0 * self.wheels

    def purchase_price(self):
        """Return the price for which we would pay to purchase the truck."""
        if self.sold_on is None:
            return 0.0  # Not yet sold
        return 10000 - (.10 * self.miles)

    ...
```

Wow. That's *almost identical* to the car class. One of the most important rules of programming (in general, not just when dealing with objects) is "DRY" or "**D**on't **R**epeat **Y**ourself. We've definitely repeated ourselves here. In fact, the `Car` and `Truck` classes differ only by *a single character* (aside from comments).

So what gives? Where did we go wrong? Our main problem is that we raced straight to the concrete: `Car` s and `Truck` s are real things, tangible objects that make intuitive sense as classes. However, they share so much data and functionality in common that it seems there must be an *abstraction* we can introduce here. Indeed there is: the notion of `Vehicle` s.

Abstract Classes

A `Vehicle` is not a real-world object. Rather, it is a *concept* that some real-world objects (like cars, trucks, and motorcycles) embody. We would like to use the fact that each of these objects can be considered a vehicle to remove repeated code. We can do that by creating a `Vehicle` class:

```
class Vehicle(object):
    """A vehicle for sale by Jeffco Car Dealership.

    Attributes:
        wheels: An integer representing the number of wheels the vehicle has.
        miles: The integral number of miles driven on the vehicle.
        make: The make of the vehicle as a string.
        model: The model of the vehicle as a string.
        year: The integral year the vehicle was built.
        sold_on: The date the vehicle was sold.
    """

    base_sale_price = 0

    def __init__(self, wheels, miles, make, model, year, sold_on):
        """Return a new Vehicle object."""
        self.wheels = wheels
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on


    def sale_price(self):
        """Return the sale price for this vehicle as a float amount."""
        if self.sold_on is not None:
            return 0.0  # Already sold
        return 5000.0 * self.wheels

    def purchase_price(self):
        """Return the price for which we would pay to purchase the vehicle."""
        if self.sold_on is None:
            return 0.0  # Not yet sold
        return self.base_sale_price - (.10 * self.miles)
```

Now we can make the `Car` and `Truck` class *inherit* from the `Vehicle` class by replacing `object` in the line `class Car(object)`. The class in parenthesis is the class that is inherited from (`object` essentially means "no inheritance". We'll discuss exactly why we write that in a bit).

We can now define `Car` and `Truck` in a very straightforward way:

```
class Car(Vehicle):

    def __init__(self, wheels, miles, make, model, year, sold_on):
        """Return a new Car object."""
        self.wheels = wheels
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on
        self.base_sale_price = 8000


class Truck(Vehicle):

    def __init__(self, wheels, miles, make, model, year, sold_on):
        """Return a new Truck object."""
        self.wheels = wheels
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on
        self.base_sale_price = 10000
```

This works, but has a few problems. First, we're still repeating a lot of code. We'd ultimately like to get rid of **all** repetition. Second, and more problematically, we've introduced the `Vehicle` class, but should we really allow people to create `Vehicle` objects (as opposed to `Car` s or `Truck` s)? A `Vehicle` is just a concept, not a real thing, so what does it mean to say the following:

```
v = Vehicle(4, 0, 'Honda', 'Accord', 2014, None)
print v.purchase_price()
```

A `Vehicle` doesn't have a `base_sale_price` , only the individual *child* classes like `Car` and `Truck` do. The issue is that `Vehicle` should really be an *Abstract Base Class*. Abstract Base Classes are classes that are only meant to be inherited from; you can't create *instance* of an ABC. That means that, if `Vehicle` is an ABC, the following is illegal:

```
v = Vehicle(4, 0, 'Honda', 'Accord', 2014, None)
```

It makes sense to disallow this, as we never meant for vehicles to be used directly. We just wanted to use it to abstract away some common data and behavior. So how do we make a class an ABC? Simple! The `abc` module contains a metaclass called `ABCMeta` (metaclasses are a bit outside the scope of this article). Setting a class's

metaclass to `ABCMeta` and making one of its methods *virtual* makes it an ABC. A *virtual* method is one that the ABC says must exist in child classes, but doesn't necessarily actually implement. For example, the Vehicle class may be defined as follows:

```
from abc import ABCMeta, abstractmethod

class Vehicle(object):
    """A vehicle for sale by Jeffco Car Dealership.


    Attributes:
        wheels: An integer representing the number of wheels the vehicle has.
        miles: The integral number of miles driven on the vehicle.
        make: The make of the vehicle as a string.
        model: The model of the vehicle as a string.
        year: The integral year the vehicle was built.
        sold_on: The date the vehicle was sold.
    """

    __metaclass__ = ABCMeta

    base_sale_price = 0

    def sale_price(self):
        """Return the sale price for this vehicle as a float amount."""
        if self.sold_on is not None:
            return 0.0  # Already sold
        return 5000.0 * self.wheels

    def purchase_price(self):
        """Return the price for which we would pay to purchase the vehicle."""
        if self.sold_on is None:
            return 0.0  # Not yet sold
        return self.base_sale_price - (.10 * self.miles)

    @abstractmethod
    def vehicle_type():
        """Return a string representing the type of vehicle this is."""
        pass
```

Now, since `vehicle_type` is an `abstractmethod`, we can't directly create an instance of `Vehicle`. As long as `Car` and `Truck` inherit from `Vehicle` **and** define `vehicle_type`, we can instantiate those classes just fine.

Returning to the repetition in our `Car` and `Truck` classes, let see if we can't remove that by hoisting up common functionality to the base class, `Vehicle`:

```
from abc import ABCMeta, abstractmethod
class Vehicle(object):
    """A vehicle for sale by Jeffco Car Dealership.


    Attributes:
        wheels: An integer representing the number of wheels the vehicle has.
        miles: The integral number of miles driven on the vehicle.
        make: The make of the vehicle as a string.
        model: The model of the vehicle as a string.
        year: The integral year the vehicle was built.
        sold_on: The date the vehicle was sold.
    """

    __metaclass__ = ABCMeta

    base_sale_price = 0
    wheels = 0

    def __init__(self, miles, make, model, year, sold_on):
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on

    def sale_price(self):
        """Return the sale price for this vehicle as a float amount."""
        if self.sold_on is not None:
            return 0.0  # Already sold
        return 5000.0 * self.wheels

    def purchase_price(self):
        """Return the price for which we would pay to purchase the vehicle."""
        if self.sold_on is None:
            return 0.0  # Not yet sold
        return self.base_sale_price - (.10 * self.miles)

    @abstractmethod
    def vehicle_type(self):
        """Return a string representing the type of vehicle this is."""
        pass
```

Now the `Car` and `Truck` classes become:

```
class Car(Vehicle):
    """A car for sale by Jeffco Car Dealership."""

    base_sale_price = 8000
    wheels = 4

    def vehicle_type(self):
        """Return a string representing the type of vehicle this is."""
        return 'car'

class Truck(Vehicle):
    """A truck for sale by Jeffco Car Dealership."""

    base_sale_price = 10000
    wheels = 4

    def vehicle_type(self):
        """Return a string representing the type of vehicle this is."""
        return 'truck'
```

This fits perfectly with our intuition: as far as our system is concerned, the only difference between a car and truck is the base sale price. Defining a `Motorcycle` class, then, is similarly simple:

```
class Motorcycle(Vehicle):
    """A motorcycle for sale by Jeffco Car Dealership."""

    base_sale_price = 4000
    wheels = 2

    def vehicle_type(self):
        """Return a string representing the type of vehicle this is."""
        return 'motorcycle'
```

Inheritance and the LSP

Even though it seems like we used inheritance to get rid of duplication, what we were *really* doing was simply providing the proper level of abstraction. And *abstraction* is the key to understanding inheritance. We've seen how one side-effect of using inheritance is that we reduce duplicated code, but what about from the *caller's perspective*. How does using inheritance change that code?

Quite a bit, it turns out. Imagine we have two classes, `Dog` and `Person`, and we want to write a function that takes either type of object and prints out whether or not the instance in question can speak (a dog can't, a person can). We might write code like the following:

```
def can_speak(animal):
    if isinstance(animal, Person):
        return True
    elif isinstance(animal, Dog):
        return False
    else:
        raise RuntimeError('Unknown animal!')
```

That works when we only have two types of animals, but what if we have twenty, or *two hundred*? That `if...elif` chain is going to get quite long.

The key insight here is that `can_speak` shouldn't care what type of animal it's dealing with, the animal class itself should tell *us* if it can speak. By introducing a common base class, `Animal`, that defines `can_speak`, we relieve the function of it's type-checking burden. Now, as long as it knows it was an `Animal` that was passed in, determining if it can speak is trivial:

```
def can_speak(animal):
    return animal.can_speak()
```

This works because `Person` and `Dog` (and whatever other classes we crate to derive from `Animal`) follow the *Liskov Substitution Principle*. This states that we should be able to use a child class (like `Person` or `Dog`) wherever a parent class (`Animal`) is expected an everything will work fine. This sounds simple, but it is the basis for a powerful concept we'll discuss in a future article: *interfaces*.

Summary

Hopefully, you've learned a lot about what Python classes are, why they're useful, and how to use them. The topic of classes and Object-oriented Programming are insanely deep. Indeed, they reach to the core of computer science. This article is not meant to be an exhaustive study of classes, nor should it be your only reference. There are literally thousands of explanations of OOP and classes available online, so if you didn't find this one suitable, certainly a bit of searching will reveal one better suited to you.

As always, corrections and arguments are welcome in the comments. Just try to keep it civil.

Lastly, it's not too late to see me speak at the upcoming Wharton Web Conference (https://www.sas.upenn.edu/wwc/) at UPenn! Check the site for info and tickets.

Posted on Jun 18, 2014 by Jeff Knupp

Tweet

Like Share 167 people like this. Sign Up to see what your friends like.

« REST APIs, ORMs, And The Neglected Client (/blog/2014/06/10/rest-apis-orms-and-the-neglected-client/)

Like this article?

Why not sign up for **Python Tutoring**? Sessions can be held remotely using Google+/Skype or in-person if you're in the NYC area. Email jeff@jeffknupp.com (mailto:jeff@jeffknupp.com) if interested.

Sign up for the free jeffknupp.com email newsletter. Sent roughly once a month, it focuses on Python programming, scalable web development, and growing your freelance consultancy. And of course, you'll never be spammed, your privacy is protected, and you can opt out at any time.

* indicates required

Email Address *

Subscribe

Copyright © 2018 - Jeff Knupp- Powered by Blug (https://www.github.com/jeffknupp/blug)

CLICKY ANALYTICS (https://clicky.com/66535137)