Jeff Knupp

PYTHON PROGRAMMER

BLOG (/)   ABOUT (/about-me)   ARCHIVES (/blog/archives)   TUTORING (/python-tutoring)
BOOK (https://jeffknupp.com/writing-idiomatic-python-ebook/)

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:

```python
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:

```python
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.

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:

```
class Car(object):

    wheels = 4

    def __init__(self, make, model):
        self.make = make
        self.model = model

mustang = Car('Ford', 'Mustang')
print mustang.wheels
# 4
print Car.wheels
# 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:

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

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:

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

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:

```python
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. *Inherticance* 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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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`:

```python
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:

```python
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:

```python
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*.
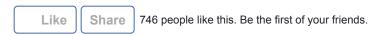
Summary

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.
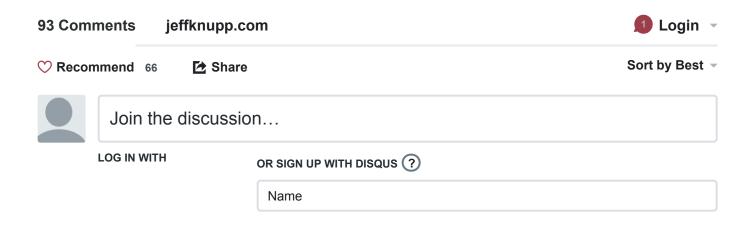
Posted on Jun 18, 2014 by Jeff Knupp

Tweet

Like   Share   746 people like this. Be the first of your friends.

« 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 (http://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

---

93 Comments   **jeffknupp.com**   🔴 1 **Login** ▾

♡ **Recommend** 66   ↪ **Share**   Sort by Best ▾

[Join the discussion…]

LOG IN WITH   **OR SIGN UP WITH DISQUS** ⑦

[Name]

Srinidhi Skanda • 2 years ago
thank you. what the advantage use of class in python where module acts as a class. i.e in java we use class variable to get fields and function. same thing with the module concept here (We can access variables or function using module name) if we can access variables by using module name tahn what the use of class in python
24 ∧ | ∨ • Reply • Share ›

Pham Huu Bang ➜ Srinidhi Skanda • a year ago
one of the use of class in Python is inheritance.
1 ∧ | ∨ • Reply • Share ›

Roger Duthie • 8 months ago

I enjoyed reading this blog post on OO in Python. I'm not an OO newbie, nor a Python newbie, but it's nice to consolidate what you know. I have one point: in post, the author states: 'object essentially means "no
inheritance". We'll discuss exactly why we write that in a bit', relating to the specification of a parent class in the class definition.

The idea that this is no inheritance is not true, although perhaps the simplification is justified in a discussion at this level. The class which states 'object' as an argument in its class definition is actually inheriting from the Python object class (am I right?).

The reason I'm trawling around reading Python blogs this morning is to get more information about the 'object' parent class. I know there are a lot of features* that this parent class has and I am looking for information on what is good to override in my own classes and what will have dire side-effects.

Anyway, thanks for the enjoyable read. I'll consider buying your book.

* the features, for example, that you see when you type: dir(object)

15 ∧ | ∨ • Reply • Share ›

stairclimber • 3 years ago

This is a great intro to OOP until you started talking about "instance of an ABC" and ABCMeta and metaclass and abstract base classes..

We newbies have no frame of reference in earlier parts of the article regarding abc .

In fact, having read a few beginner Python books and videos, this is the first time I hear about them.

I wonder how useful ABCMeta is compared to concepts such as decorators which I have come across a few times in my reading for a newbie .

I think you need to break up this article and write another OOP article introducing them to dummies.

Here in this article you moved to the deep end too quickly for newbies like me.

I hope this article is not an indication of the quality of your book , i.e. getting too deep end too quickly and presuming lots of knowledge on the parts of newbies.

22 ∧ | ∨ • Reply • Share ›

Ken Lin ↱ stairclimber • 2 years ago

I disagree; This is the first article about Python Classes (or classes in any programming language) that I have ever read, and I thought Jeff wrote it with beginners in mind. If it helps, try re-reading the article a second time and a lot of your questions might be cleared up. I definitely recommend doing this when learning a new concept like classes and OOP

concept like classes and OOP.

Another reason why you should re-read the article because your complaint ""This is a great intro to OOP until you started talking about "instance of an ABC" and ABCMeta and metaclass and abstract base classes.." is clearly addressed by Jeff:

Here I list the new terms introduced in the article and the explanation offered in the article:

Term: Abstract Base Classes
Explanation: "Abstract Base Classes" are classes that are only meant to be inherited from; you can't create instance of an ABC."
Note: If you have been following the article, you should have no trouble understanding what a class is. For the purpose of understanding the examples, you only need to treat ABC as a class that can not be instantiated (as described in the quote, duh).

Term: Metaclass/ABCMeta
Explanation: "Setting a class's metaclass to ABCMeta and making one of its methods virtual makes it an ABC."
Note: Metaclass was briefly mentioned for the sake of ABC. You only need to know that two conditions must be met in order to create an ABC, and one condition is setting __metaclass__ to ABCmeta. No further discussion regarding metaclass is present because "metaclasses are a bit outside the scope of this article."

5 ∧ | ∨ • Reply • Share ›

Trip Kendall ➜ Ken Lin • 6 months ago
Good call - I read it more than once and got something out of it the 2nd time as well

5 ∧ | ∨ • Reply • Share ›

Jeff Silverman ➜ Ken Lin • a year ago
Ken, my experience in both teaching programming and in mountain search and rescue is that if you are going to rescue somebody lost in the mountains, then you have to go to where that person is before you can help him or her. You can't tell somebody who is lost to simply walk North, because if they are lost, then they might not know which way North is.

I agree with what you wrote: Jeff did a nice essay. I also agree with stairclimber that he got lost at the Abstract Base Class point. Maybe for stairclimber's first trip into the mountains, he should not tackle Mt. Rainier, maybe he should try Pike's peak. Yes, definitely climb mountains, just get a little experience under your belt.

When I am writing in Java, then yes, I use abstractions. In Python, I use inheritance of methods which fail catastrophically if not overridden.

(Anybody replying to this message should disambiguate between Jeff Silverman and Jeff Knupp).

3 ∧ | ∨ • Reply • Share ›

Hamza Ahmed → Ken Lin • 6 days ago

I am a beginner in regards to OOP and i have found this article an incredible gateway to understanding OOP . As Jeff says , it's not meant to be the only reference.

For me, i'm going through at least 2-3 online resources jumping back and forth progressively and i have found my self consistently circling back to this article to use as a benchmark - just to see how much my understanding has increased.

And it has been a treat.

I have re-read this maybe 3-4 times now and can say i can follow all the ideas in here much much sooner than i should be.

Consider this read, if you a newbie, as a pre-read to help you get an overall grasp on the power of classes so you may get a sense of direction in your learning.

∧ | ∨ • Reply • Share ›

Jeff Silverman → stairclimber • a year ago

There is a another way to make a class effectively an abstract class: put a method in it that only raises a NotImplemented or NotImplementedError exception. If any class inherits from this class, and does *not* override the method definition, then when an object is instantiated from this class and invokes the method, the NotImplemented exception will be raised. On the other hand, if a class inherits from this class, and *does* override the method, then the overriding method will be invoked and it will (presumably) yield a correct answer.

1 ∧ | ∨ • Reply • Share ›

stairclimber → Jeff Silverman • a year ago

Thanks Mr Jeff Silverman for your support. I love your analogy of mountain search and rescue. Pretty enlightening.

25 ∧ | ∨ • Reply • Share ›

sravanthi → stairclimber • 4 months ago

as a beginner in programming, I could get the content after re-reading the same article . Wonderful explanation ... thank you

∧ | ∨ • Reply • Share ›

Mahendra Singh • 2 years ago

excellent crystal clear

5 ∧ | ∨ • Reply • Share ›

Dasagriva Manu • a year ago

Very comprehensive post on OOP concepts in Python. But as I've learnt that Python doesn't

very comprehensive post on OOP concepts in Python. But as I've learnt that Python doesn't support strong encapsulation. Then why does it use name mangling for?

2 ⌃ | ⌄ • Reply • Share ›

Jason Haas • 3 years ago

Fantastic article as usual Jeff! One comment about when you define the Car(Vehicle) and Truck(Vehicle) classes. In the Vehicle class, you have an attribute called base_sale_price which is set to 0 in the Vehicle class and then you override it in the Car and Truck classes. How does the programmer know that this needs to be implemented? Maybe it would be better suited to be part of an __init__ method? This way it has to be passed it as an argument to the Class to instantiate an object.

3 ⌃ | ⌄ • Reply • Share ›

Ken Lin ➜ Jason Haas • 2 years ago

Disclaimer that I am not nearly as experienced as Jeff when it comes to OOP or Python in general; I welcome everyone to correct me if I'm wrong or unclear

In Jeff's code, base_sale_price (and wheels, too) is set to 0 in the Vehicle abstract base class and then the Car/Truck class will "automatically" override the base_sale_price. "Automatically" in the sense that the user does not need to worry about manually inputting this value as an argument, because the value of base_sale_price is already in the code. In essence, since base_sale_price for a car/truck is always the same, it would be redundant to ask the user to pass this price as an argument during every object instantiation.

Your suggestion of putting base_sale_price in the __init__method would most certainly work, but it might be an example of what Jeff calls "Don't Repeat Yourself".

3 ⌃ | ⌄ • Reply • Share ›

Derek Thomas ➜ Ken Lin • 2 years ago

Given that no vehicle sells for: base_sale_price = 0
Is there a way to use an @abstractproperty or something like @abstractmethod but for properties?

⌃ | ⌄ • Reply • Share ›

Anthony Tirop ➜ Ken Lin • 2 years ago

Woow, **@Ken Lin** I like your explanation approach, Jeff's article has now sunk. Actually I was reasoning out as **@Jason Haas** only to realize I was already treating the child classes as the object instances, that way Jason's/mine thinking works. Thank you. I am an intermediate pythonister.

⌃ | ⌄ • Reply • Share ›

Toussah • 2 years ago

So... is nobody gonna make a comment about that 4-wheeled truck?

2 ⌃ | ⌄ • Reply • Share ›

Mark du Preez ➜ Toussah • a year ago

;-D

^ | ∨ • Reply • Share ›

Knittingarch → Toussah • 2 years ago

Hahaha! I tried to work ahead without looking at the answers, and I was definitely surprised by that! I guessed 8-wheeler, but maybe it's supposed to be four pairs?

^ | ∨ • Reply • Share ›

jeffknupp Jeff → Knittingarch • 2 years ago

Truck != huge freight shipping truck. A Ford F-150 is a truck with four wheels, for example

6 ^ | ∨ • Reply • Share ›

Jeff Silverman • a year ago

Minor typographical error: Inherticance should be "inheritance". I think that's what Scott Broscious was trying to tell you a year ago.

Also, "perhaps we should crate a Truck" should be "perhaps we should create a Truck".

1 ^ | ∨ • Reply • Share ›

papaKenya • 2 years ago

I understood everything except the past part - can_speak(animal). Is animal here the base class? I didn't understand this section.

1 ^ | ∨ • Reply • Share ›

Mark du Preez → papaKenya • a year ago

Yes, it is.

^ | ∨ • Reply • Share ›

Paulo Raposo • 3 years ago

Super useful and nicely written - thanks :)

1 ^ | ∨ • Reply • Share ›

Zach N • 3 years ago

When using your "Inheritance" example I find that my IDE gives a warning, "Call to __init__ of super class is missed." Is using a super class necessary or 'better' in this instance?
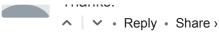
1 ^ | ∨ • Reply • Share ›

Ankush Thakur → Zach N • 2 years ago

Neither. Super call __init__ methods are called only if they serve a useful purpose. It's up to you how you design your classes. I know this answer doesn't help, but unfortunately there's no straightforward answer.

^ | ∨ • Reply • Share ›

Felix • 19 days ago

Thanks!

~ | ⌄ • Reply • Share ›

Ozs • 2 months ago

Thank you very much!

:)

~ | ⌄ • Reply • Share ›

adnan virk • 2 months ago

great explanation

~ | ⌄ • Reply • Share ›

Robert Raducanu • 2 months ago

This is a great OOP starter tutorial. I've read a some books from APress, PACKT etc with different programming languages but this is certainly the BEST tutorial and could outshine any other chapters describing OOP I've ever read. Good job!

~ | ⌄ • Reply • Share ›

Ivan Abraham • 2 months ago

"flat rate - 10% of the miles driven on the vehicle"

Minus Jeff minus ... I spent a good five minutes trying to figure this one out because I thought it was hyphen, then gave up and moved and then realized in the code that its a minus.

~ | ⌄ • Reply • Share ›

Maux Boyle • 2 months ago

Thanks for this. I'm working through an online Python class and got rather hung up on classes. This post helped me wrap my head around them a little better.

~ | ⌄ • Reply • Share ›

Franklin B. Herman • 3 months ago

Excellent introduction to OOP. I especially appreciated the ABC examples.

~ | ⌄ • Reply • Share ›

pradip gareja • 3 months ago

GREAT ....SIR and thaks

~ | ⌄ • Reply • Share ›

Franek • 3 months ago

A great article, thanks!

~ | ⌄ • Reply • Share ›

Helmut • 3 months ago

I like the way you explain stuff. Thanks for taking the time to explain this to random people on the internet.

~ | ⌄ • Reply • Share ›

Ankit Kumar Honey • 5 months ago

Good Article about python and OOPS concept. I really enjoyed reading it. Thank you Jeff

∧ | ∨ • Reply • Share ›

Trip Kendall • 6 months ago

Hey Now! Thanks for a great article. So much of what is out there is of dubious value, but this is gold. Cheers.

∧ | ∨ • Reply • Share ›

Anzaemon • 7 months ago

I have tried to figure out classes before, by a book and internet articles but never seemed to actually grasp it. Always got lost in abstract concepts of instance and inheritance.

I think this was a very enlightening article on classes on a basic level, and the picture cleared for me. Perhaps it was easier for me because I had some understanding of the vocabulary.

One thing though, I had some issues trying these codes out, until I realised that there should be brackets after print.

But still though, a great article!

∧ | ∨ • Reply • Share ›

nathan woodard • 7 months ago

Thanks for posting. Really great! :)

∧ | ∨ • Reply • Share ›

ibQ • 9 months ago

It would seem fitting that if not ABCing the vehicle class, then vehicle_type would be implemented as a class attribute along with wheels and base_sale_price.

So in order to make vehicle uninstantiable, I've got to turn what would logically be an attribute into a method? That seems like a hack.

Now, I realize that this may just be an overly simple example from which I am obtusely drawing a wrong conclusion - but with no explanation, and the code being what it is, is that a relevant take? IOW, lacking a suitable method, but wishing to ABC a class, is this, or just providing a dummy (pass) method in the child, a common, accepted practice?

∧ | ∨ • Reply • Share ›

Daniel Zuzevich • a year ago

Good stuff. Thanks for this.

∧ | ∨ • Reply • Share ›

Abdalla AVmed • a year ago

Thank you very much. Really I didn't understand OOP as much as now after reading this article

˄ | ˅ • Reply • Share ›

John Hanly • a year ago

How do you get the sale_price or purchase_price of a car. I did following
my_car = Car(23000, 'Ford', 'Taurus', 2015, 2016)
print(my_car.make, my_car.miles) 'Ford', 23000

I found my mistake.

print(my_car.sale_price()) I forgot the ( )...

˄ | ˅ • Reply • Share ›

Sam Parson • a year ago

This is good for basic oop, but where does inheritance and polymorphism come into play?
For example, java can use key words like "extend", "implement", "super", "instanceof", and
allow Override for methods of other classes...

˄ | ˅ • Reply • Share ›

Peter M. Peters • a year ago

ok.. i get the logic behind the abstract base classes and while you said it renders it illegal to
initialise a direct object of the vehicle class, i wonder how it enforces such kind of an
argument given that when i created a vehicle object "v" and ran the v.sale_price(); the code
executed just fine. in truth, i was expecting some error or warning which would have shown
that adding the metaclass definition was not just typing practice

˄ | ˅ • Reply • Share ›

eachhisownchimera ➜ Peter M. Peters • a year ago

Same question!!
Found out that the above code stops instantiation in Python 2 but in Python 3 it does
not. For Python 3, you can do:

from abc import ABCMeta, abstractmethod
class Abstract(metaclass=ABCMeta):
@abstractmethod
def foo(self):
pass

a = Abstract()
----> TypeError: Can't instantiate abstract class Abstract with abstract methods foo

https://stackoverflow.com/q...

˄ | ˅ • Reply • Share ›

Gaurav Gupta • a year ago

Great Article

˄ | ˅ • Reply • Share ›

Charl Swart • a year ago

This is an awesome post! One of the best describing classes and methods. Been coding in Python for a year or so, done some very cool stuff, but I need to make my code more compact, and this showed me where I can.

∧ | ∨ • Reply • Share ›

Load more comments

ALSO ON JEFFKNUPP.COM

Counting Cards With Python
1 comment • 6 months ago

Avatar hfuhruhurr — Sweet read. I remember getting hardcore about analyzing blackjack when I lived in Vegas.Regarding the +1/-1 …

Why I Hate HATEOAS
8 comments • 3 years ago

Avatar cblin — Sorry to revive your old post but it did come first when I type "hateoas problem" in google :)I am no HATEOAS or REST …

Python, sandman2, and Open Data
1 comment • 2 years ago

Avatar Michael Aye — FYI, your sandman2 link points to sandman.

Improve Your Python: the with Statement and Context Managers
1 comment • 2 years ago

Avatar Jansen — great article, you do talk about wierd parts of python

✉ Subscribe   Ⓓ Add Disqus to your siteAdd DisqusAdd   🔒 Disqus' Privacy PolicyPrivacy PolicyPrivacy