tuts+

PYTHON

Python from Scratch: Object Oriented Programming

by Giles Lavelle *24 Aug 2011* *85 Comments*

This post is part of a series called Python from Scratch.

Python from Scratch - Functions and Modules

Python from Scratch - Create a Dynamic Website

Welcome back to lesson four in our *Python from Scratch* series. This tutorial will assume some prior knowledge of variables, data types, functions and print output. If you're not up to date, check out the previous three articles in the series to catch up.

Today, we're going to be delving into the subject of Object Oriented Programming (OOP). OOP is a very powerful way of organizing your code, and a solid understanding of the concepts behind it can really help you get the most out of your coding.

Prefer a Screencast?

Python from Scratch - Lesson 4: Object Oriented Programming

Python from Scratch - Lesson 4: Object Oriented Programming

Transcription

What is Object Oriented Programming?

Python is primarily designed as an object-oriented programming language – but what does ‘object oriented’ actually mean?

There are a variety of definitions for the term, and you could talk for literally hours trying to explain the complicated ins and outs, nuances and differences in implementations, but I'll try to give a quick overview.

Broadly, object oriented programming is the concept that, in programming, the objects that we're manipulating are more important than the logic needed to manipulate those objects. Traditionally, a program has been seen as a recipe – a set of instructions that you follow from start to finish in order to complete a task. That can still be true, and for many simple programs, that's all which is required. That approach is sometimes known as procedural programming.

> *OOP puts objects at the center of the process.*

On the other hand, as programs get more and more complex and convoluted, the logic needed to write them in a purely procedural way gets more and more twisted and hard to understand. Often object oriented approaches can help with that.

When we talk about object oriented approaches, what we do is put the objects at the center of the process, instead of simply using them as necessary containers for information as part of our procedural instructions. First, we define the objects we want to manipulate and how they relate to each other, and then we start to flesh it out with logic to make the program actually work.

When I talk about 'objects', I can be talking about all sorts of things. An 'object' can represent a person (as defined by properties such as name, age, address etc.), or a company (as defined by things like number of employees and so on), or even something much more abstract, like a button in a computer interface.

In this introduction, we're not going to be covering all of the concepts in this topic because we'd be here all night, but by the end of the tutorial, I hope you'll have a solid understanding of the principles you need to start straight away using some simple object-oriented techniques in your Python programs. Even better, these concepts are fairly similar in a lot of programming environments. The knowledge transfers over from language to language quite nicely.

Getting Started

I mentioned earlier that the first thing we should do when we're going for an OOP approach is to define the objects we're going to be using. The way we do this is to first define the properties that it possesses using a class. You can think of a class as a sort of template; a guide for the way an object should be structured. Each object belongs to a class and inherits the properties of that class, but acts individually to the other objects of that class.

> *An object is sometimes referred to as an 'instance' of a class.*

As a simple example, you might have a class named 'person' with, say, an age and a name property, and an instance of that class (an object) would be a single person. That person might have a name of "Andy" and an age of 23, but you could simultaneously have another person belonging to the same class with the name of "Lucy" and an age of 18.

It's hard to understand this without seeing it in practice, so let's get some actual code going.

Defining a class

To define a class, in typical simple Python fashion, we use the word 'class,' followed by the name of your new class. I'm going to make a new class here, called 'pet'. We use a colon after the name, and then anything contained within the class definition is indented. However, with a class, there are no parentheses:

```
class pet:
```

So now we've got a class, but it's rather useless without anything in it. To start, let's give it a couple of properties. To do this, you simply define some variables inside the class – I'm going to go with the number of legs to start with. As usual, you should always name your variables so that it's easy to tell what they are. Let's be original and call it 'number_of_legs'. We need to define a value or we'll get an error. I'll use 0 here (it doesn't matter too much in this case since the number of legs will be specific

to each instance of the class - a fish doesn't have the same amount of legs as a dog or a duck, etc. - so we'll have to change that value for each object anyway).

```
class pet:
        number_of_legs = 0
```

Instances and member variables

A class on its own isn't something you can directly manipulate; first, we have to create an instance of the class to play with. We can store that instance in a variable. Outside of the class (without any indentation), let's make an instance of the class and store it in the variable, 'doug'. To make a new instance of a class, you simply type the name of the class, and then a pair of parentheses. At this point, there's no need to worry about the parentheses, but later on you'll see that they're there because, like a function, there's a way of passing in a variable for use by the class when you first create the instance.

> *A class on its own isn't something that you can directly manipulate.*

```
class pet:
        number_of_legs = 0

doug = pet()
```

Now that we have an instance of a class, how do we access and manipulate its properties? To reference a property of an object, first we have to tell Python which object (or which instance of a class) we're talking about, so we're going to start with 'doug'. Then, we're going to write a period to indicate that we're referencing something that's contained within our doug instance. After the period, we add the name of our variable. If we're accessing the `number_of_legs` variable, it's going to look like this:

```
doug.number_of_legs
```

We can treat that now exactly as we would treat any other variable – here I'm going to assume doug is a dog, and will give that variable the value of 4.

To access this variable, we're going to use it again exactly as we would treat any other variable, but using that `doug.number_of_legs` property instead of the normal variable name. Let's put in a line to print out how many legs doug has so that we can show that it's working as it should:

```
class pet:
        number_of_legs = 0

doug = pet()
doug.number_of_legs = 4
print "Doug has %s legs." % doug.number_of_legs
```

If you run the code above, you'll see that it's printed out for us. It defined our 'pet' class, created a new instance of that class and stored it in the variable 'doug', and then, inside that instance, it's assigned the value of 4 to the `number_of_legs` variable that it inherited from its class.

So you can see from that very simplified example how you can begin to build nice, modular data structures that are clear and easy to use, and can start to scale quite nicely.

Introducing Logic

Okay, so that's the very basics of classes and objects, but at the moment we can only really use classes as data structures - or, containers for variables. That's all well and good, but if we want to start performing more complex tasks with the data we're manipulating, we need a way of introducing some logic into these objects. The way we do that is with methods.

Methods, essentially, are functions contained within a class. You define one in exactly the same way as you would a function, but the difference is that you put it

inside a class, and it belongs to that class. If you ever want to call that method, you have to reference an object of that class first, just like the variables we were looking at previously.

> *Methods, essentially, are functions contained within a class.*

I'm going to write a quick example here into our pet class to demonstrate; let's create a method, called 'sleep', which is going to print out a message when it's first called. Just like a function, I'm going to put 'def' for 'define', and then I'm going to write the name of the method I want to create. Then we're going to put our parentheses and semicolon, and then start a new line. As usual, anything included in this method is going to be indented an extra level.

Now, there is another difference between a method and a function: a method always, always, always has to have an argument, called 'self' between the parentheses. When Python calls a method, what it does is passes the current object to that method as the first argument. In other words, when we call `doug.sleep()`, Python is actually going to pass the object 'doug' as an argument to the sleep method.

We'll see why that is later, but for now you need to know that, with a method, you always have to include an argument called 'self' first in the list (if you want to add more arguments, you can add them afterwards, exactly like if you were passing multiple arguments to a function). If you don't include that argument, when you run the code, you're going to get an error thrown because Python is passing in an argument (this 'self' object), and the method is saying, 'Hey, man, I don't take any arguments, what are you talking about?'. It's the same as if you tried to pass an argument into a function that doesn't accept any arguments.

So here's what we have so far:

```
class pet:
        number_of_legs = 0

        def sleep(self):
```

```
doug = pet()
```

Inside this method, we're going to write a print statement like so:

```
class pet:
        number_of_legs = 0

        def sleep(self):
                print "zzz"


doug = pet()
```

Now, if we want to use this method, we simply use an instance of the pet class to reference it. Just like the `number_of_legs` variable, we write the name of the instance (we've got one called doug), then a period, then the name of the method including parentheses. Note that we're calling sleep using no arguments, but Python is going to add in that argument by itself, so we're going to end up with the right amount of arguments in total.

```
class pet:
        number_of_legs = 0

        def sleep(self):
                print "zzz"


doug = pet()
doug.sleep()
```

If you run this code, you should see that it prints out the message we wrote.

Data

Great, so now how about we write a new method to print out how many legs the pet has, to demonstrate how you can use methods to start manipulating the data within the class, and to demonstrate why we need to include this confusing 'self' argument. Let's make a new method, called '`count_legs`'.

This is where the 'self' argument comes in. Remember when we were accessing `number_of_legs` from outside the class and we had to use 'doug.number_of_legs' instead of just 'number_of_legs'? The same principle applies; if we want to know what is contained in that variable, we have to reference it by first specifying the instance containing that variable.

However, we don't know what the instance is going to be called when we write the class, so we get around that using the 'self' variable. 'self' is just a reference to the object that is currently being manipulated. So to access a variable in the current class, you simply need to preface it with 'self' and then a period, like so:

```
class pet:
        number_of_legs = 0

        def sleep(self):
                print "zzz"

        def count_legs(self):
```

```
                print "I have %s legs" % self.number_of


doug = pet()
doug.number_of_legs = 4
doug.count_legs()
```

In practice, what this means is that wherever you write ‘self’ in your method, when you run the method that self is replaced by the name of the object, so when we call ‘doug.count_legs()’ the ‘self’ is replaced by ‘doug’. To demonstrate how this works with multiple instances, let's add a second instance, representing another pet, called 'nemo':

```
class pet:
        number_of_legs = 0

        def sleep(self):
                print "zzz"

        def count_legs(self):
                print "I have %s legs" % self.number_of


doug = pet()
doug.number_of_legs = 4
doug.count_legs()

nemo = pet()
nemo.number_of_legs = 0
nemo.count_legs()
```

This will print out a message for 4 and then 0 legs, just as we wanted, because when we call ‘nemo.count_legs(),’ the ‘self’ is replaced by ‘nemo’ instead of 'doug'.

In this way, our method will run exactly as intended because the ‘self’ reference will

dynamically change depending on the context and allow us to manipulate the data only within the current object.

The main things you need to remember about methods is that they're exactly like functions, except that the first argument has to be 'self' and that to reference an internal variable you have to preface the variable name with 'self'.

Just as a note: You can actually use any name instead of 'self' for your methods. – The methods here would work just as well if we renamed the variable 'self' to any word. Using the name 'self' is simply a convention which is useful to Python programmers because it makes the code much more standard and easy to understand, even if it's written by someone else. My advice would be to stick to the conventions.

Some More Advanced Features

Now that we've gone over the basics, let's have a look at some more advanced features of classes, and how they can help make your programming easier to structure.

The next thing we're going to talk about is inheritance. As its name might hint, inheritance is the process of making a new class based around a parent class, and allowing the new class to inherit the features of the parent class. The new class can take all of the methods and variables from the parent class (often called the 'base' class).

> *Inheritance is the process of making a new class based around a parent class.*

Let's extend our pet example to see how this might be useful. If we use 'pet' as our parent class, we could create a child class which inherited from the pet class. The child class might be something like 'dog', or 'fish' – something that is still a 'pet', but is more specific than that. A dog is a pet, and does the same things that all pets do – for example it eats and sleeps and has an age and a number of legs – but it does other things that are specific to being a dog, or at least more specific than to being a

pet: dogs have fur, but not all pets do. A dog might bark, or fetch a stick, but not all pets would.

Getting back to the point, say we wanted to make a class in our program to represent a dog. We could use inheritance to inherit the methods and variables contained in 'pets' so that our dog could have a 'numberOf Legs' and the ability to 'sleep', in addition to all the dog specific things we might store or do.

Now, you might be wondering why we don't put those methods and variables into the dog class and get rid of the pet class entirely? Well, inheritance gives us two distinct advantages over that approach: One, if we want an object that is a pet, but isn't a dog – a generic pet, if you will – we can still do that. Two, perhaps later we want to add a second type of pet – maybe a fish. We can make that second class also inherit from pet, and so both classes can share everything in pet, but at the same time have their own more specific methods and variables that apply only to that type of object.

We're getting a little bogged down in the theory here, so let's write something to make it a little clearer. First, we're going to write a new class, called 'dog', but this time, between the class name and the colon, we're going to put some parentheses, and in them, we're going to write the name of the class that we want to inherit from, sort of as if we're passing this new class an argument, like we would a function.

Next, let's give this class a simple method to demonstrate how it works. I'm going to add a '`bark`' method which will print 'woof':

```
class pet:
        number_of_legs = 0

        def sleep(self):
                print "zzz"

        def count_legs(self):
                print "I have %s legs" % self.number_of

class dog(pet):
```

```
        def bark(self):
                print "Woof"
```

So, now let's see what happens if we make an instance of this class. I''m going to call our new dog 'doug' again. Now, if we call `doug.bark()`:

```
class pet:
        number_of_legs = 0

        def sleep(self):
                print "zzz"

        def count_legs(self):
                print "I have %s legs" % self.number_of

class dog(pet):
        def bark(self):
                print "Woof"

doug = dog()
doug.bark()
```

As expected, doug barks. That's great, but we haven't seen anything new yet - just a class with a method. What inheritance has done for us, though, is to make all of the pet functions and variables available to us through our 'doug' object, so if I do something like this:

```
class pet:
        number_of_legs = 0

        def sleep(self):
                print "zzz"

        def count_legs(self):
                print "I have %s legs" % self.number_of
```

```
class dog(pet):
        def bark(self):
                print "Woof"

doug = dog()
doug.sleep()
```

Then the sleep method will also execute correctly. In effect, our doug object belongs to both the 'pet' AND the 'dog' class. To ensure that the variables do the same as the methods, let's try this:

```
class pet:
        number_of_legs = 0

        def sleep(self):
                print "zzz"

        def count_legs(self):
                print "I have %s legs" % self.number_of

class dog(pet):
        def bark(self):
                print "Woof"

doug = dog()
doug.number_of_legs = 4
doug.count_legs()
```

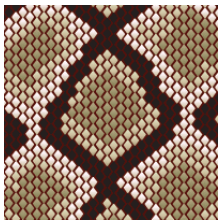
You can see that doug acts exactly as before, showing that our variables are being inherited. Our new child class is simply a specialized version of the parent one, with some extra functionality but retaining all of the previous functionality.

So there you have it, a quick introduction to object oriented programming. Stay tuned for the next installment in this series, where we're going to be working with

Python on the web!

Categories:

Python **Web Development** **OOP** **Object-Oriented Programming**



Translations:

Envato Tuts+ tutorials are translated into other languages by our community members—you can be involved too!

Translate this post Powered by native

About Giles Lavelle

N/A

Suggested Envato Tuts+ Course

Introduction to Python $9

Related Tutorials

A Smooth Refresher on Python's Modules

Code

A Smooth Refresher on Python's Classes and Objects

Code

A Smooth Refresher on Python's Functions

Code

Envato Market Item

What Would You Like to Learn?

Suggest an idea to the content editorial team at Envato Tuts+.

85 Comments Nettuts+ 1 Login

Recommend 7 Share Sort by Best

Join the discussion…

Lord Roke · 3 years ago

Fantastic tutorial. I've learned a lot from going through this. I have been learning python on and off for a few months and hit a road block when it came to OOP. But this really helps. Appriciate the time and effort taken to put this together. Thanks

36 · Reply · Share ›

Dragisa · 3 years ago

Hi Giles.
This is fantastic, I need to agree with other similar comments. I started learning Python too, and then hit the wall when classes came. Now I think I got it.

By the way, what about the "__init__" method?
Have you considered writing some info on "__init__"?
Thank you.

19 · Reply · Share ›

Jay · 3 years ago

Man! This is very nice! I was banging my head trying to understand OOP with so many sources that only confused me more until I found this page. Finally, the cloud is clearing for me!

6 · Reply · Share ›

satisfiedreader24 · 3 years ago

Just another satisfied reader wanting to say thank you!

5 · Reply · Share ›

dfsgdfg · 2 years ago

gf
df
gd
f
dfg
d
f
df
gd
fg
df
g

2 · Reply · Share ›

fh ↗ dfsgdfg · 2 years ago

gfhfg

1 · Reply · Share ›

penis ↗ dfsgdfg · a year ago

i lik u

· Reply · Share ›

pique242 · 3 years ago

Was worth reading!! Just teach in python 3+ from now on, since any one hardly uses it..

3 · Reply · Share ›

MH · 3 years ago

Thanks for writing this ! Simple and very useful for new programmers.

1 · Reply · Share ›

ivan · 3 years ago

this was great intro for beginners! infinite thanks!

1 · Reply · Share ›

Alex · 5 years ago

You might have mentioned that methods in a class support chaining by returning the class itself. Also makes python look alittle like jquery.(see last line)

```
class pet:
number_of_legs = 0

def sleep(self):
print "zzz "
return self

def set_Legs(self, legs):
self.number_of_legs = legs
return self

def count_legs(self):
print "I have %s legs" % self.number_of_legs
return self

class dog(pet):
def bark(self):
print "Woof"

doug = dog()
doug.set_Legs(4).count_legs().sleep().sleep().sleep()
```

1 · Reply · Share ›

rahul ➔ Alex · 3 years ago

Alex also please help understand the meaning of

"class dog(pet): "

· Reply · Share ›

rahul ➔ rahul · 3 years ago

Ohh my bad. Its inheritence !!!!

• Reply • Share ›

rahul Alex • 3 years ago

Hi Alex.Thanks for the additions. I was trying your code after this tutorial.
Could you please explain what benefit does adding "return self" at the end of each method mean?

• Reply • Share ›

Giles Lavelle Alex • 5 years ago

Oh yeah, that's a good point. I guess I didn't want to add too much, to try and keep in simple for beginners, but this still would have been a nice thing to include in the lesson.
Thanks for the feedback!

• Reply • Share ›

Shahad Alrawi • 7 days ago

Thank you, you have saved me from headache

• Reply • Share ›

Kapil Lohakare • 2 months ago

Awesome Tutorial. Giles Lavelle always teaches in a nice way. One correction, after 20:44, there is no video.
Thanks for great work!!!

• Reply • Share ›

Koushik Khan • 5 months ago

Really awesome! Much better than many Python books. Keep it up n I'm waiting for next post.

• Reply • Share ›

Relentlesz • a year ago

Man, this was awesome! I feel so much more confident now with learning OOP in python. THANKS A BUNCH!

• Reply • Share ›

magaka • a year ago

Thank you soooo much. This was really helpful. I was starting to feel stupid for not understanding classes...

• Reply • Share ›

Joseph • 2 years ago

Perfect

Perfect

  • Reply • Share ›

Jason Smith • 2 years ago

Error. When describing methods and how to make one, you say put a semicolon after the parenthesis...it should be a colon like so:

def typo():

I'm also curious why you put an 8 space indentation on all your examples instead of 4?

Nice tute btw

  • Reply • Share ›

John • 2 years ago

Amazing screencast - thanks!

  • Reply • Share ›

New@this_thing • 2 years ago

Thank you so much for this. I have looked all over for a simple and concise description of OOP with classes/ objects specifically for python. I almost gave up... Thanks!

  • Reply • Share ›

Pradipta Ghosh • 2 years ago

Another happy visitor. After digging around the net to understand OOP, I finally found my answer here. Very well explained for folks new to OOP.

  • Reply • Share ›

Cyberpaws • 2 years ago

Great intro to OOP Python. Well-picked examples and language for this newcomer.

  • Reply • Share ›

iheartnes • 2 years ago

Thank you very much!!!
This is really helpful and well structured information that helped me to finally understand what is going on :)

  • Reply • Share ›

Phương Nguyễn • 2 years ago

Nice tutorial. Many thanks.

  • Reply • Share ›

guest • 2 years ago

Thank you so much!!!! Your tutorial explained it so much better than my professor!

Siavash Ghomayshi · 2 years ago

a lot thanks to your excellent tutorial .....!

rahul · 3 years ago

Super kuwaaai !!!! This is what a tutorial should be like. The best part was the example (pets->dog-> number_of_legs) as it merged with the class->instance definition and thus easily understood. Also you kept only the necessary stuff and not the text book Blah blahs. Superb !!!!

Expecting more like this on python from you. What an electric start you gave me on OOPS with python.

Hulksmashchop Martin · 3 years ago

Fantastic tut!

thomas · 3 years ago

brilliant. one of the best tutorials on the web for OO principles - which is often explained poorly or overly academically.

Andrew Graham · 3 years ago

Nicely laid out examples and well worded explanations. I've learned a lot! Thank you

Kenneth Kinyanjui · 3 years ago

Very good explanation of OOP in Python

Arman · 3 years ago

That's exactly what I was looking for! Fast , simple , hands on OOP in Python. If someone is familiar with programming this is the best dive into python oop tutorial.

bamara gbon Coulibaly · 3 years ago

good tutorials , worth reading

good tutorials , worth reading

Harry • 3 years ago

Excellent tutorial, thank you!

Roopesh • 3 years ago

Really nice and concise tutorial. Explains the basics of OOP in such a simple way and in 5 minutes, so that any literate person can understand.

Sinthujan Rajendram • 3 years ago

Hi,

Thanks for the tutorial. Very well explained. can i get your email by any chance if I have any questions to clarify? I really like the way you explain things.

Erico • 3 years ago

Awesome

angie • 3 years ago

Beautiful! Many thanks :)

mike • 3 years ago

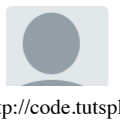
I think I haven't encountered a better explanation of the whole 'self' business than yours. Thanks!

satisfiedreder25 • 3 years ago

that was a great tutorial if only i would have had that last week my life would have been a whole lot easier

Someguy • 3 years ago

This was outstanding. Really helped me understand something that I had been struggling with. Thank you.



Jey Ell • 3 years ago

Simply Great! You are blessed, you explain OOS so easy that everybody can understand

without any problem. With all the material you have, you should make a book, I am sure in a couple of hours it will be sold out. I've watched several videos about OOS and read several books as well, your lectures are second to none, congratulations!

Reply • Share ›

Jake • 3 years ago

Great tutorial. I read straight through just once and now have a much better understanding of OOP. Please consider adding more Python OOP walkthroughs as this was a fantastic learning experience. A+

Reply • Share ›

Mika • 3 years ago

Great. Thank you!!!

Reply • Share ›

Nehal J. Wani • 3 years ago

To learn how to implement object oriented concepts in scenarios like developing a game, one can refer to the tutorial: https://www.youtube.com/watch?...

Reply • Share ›

Khurram • 3 years ago

they should have made 'self' work like 'this'(i.e shoudnt contain in the signature). Other than that I found this tutorial to be the best! Thanks alot.

Reply • Share ›