

Objects

Introduction to Objects and how they are created in Python.

We are going to look at one more Python idea called *objects*. Objects are similar to the reusable functions because we define them once and use them many times. But objects can do a lot more than simple functions can. The easiest way to understand objects is to see them in action, rather than talk loads about an abstract concept. Have a look at the following code:

```
1 # class for a dog object
2 class Dog:
3
4     # dogs can bark()
5     def bark(self):
6         print('woof!')
7         pass
8
9     pass
```

Let's start with what we're familiar with first. You can see there is a function called `bark()` inside that code. It's not difficult to see that if we called the function, we would get "woof!" printed out. That's easy enough.

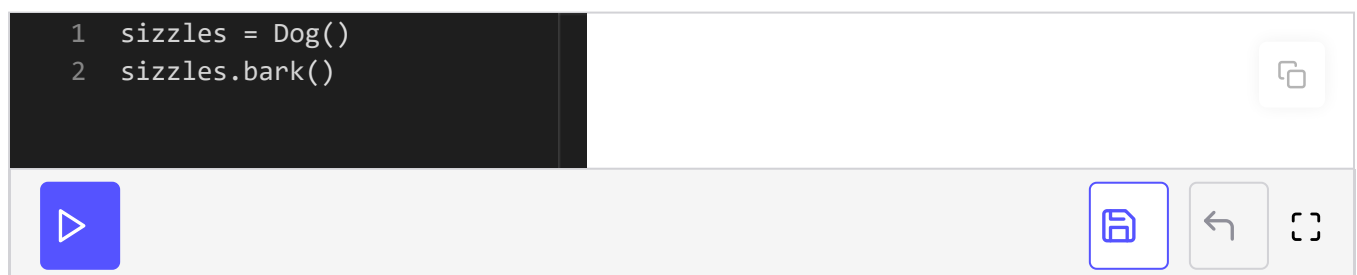
Let's look at that familiar function definition now. You can see the `class` keyword, and a name `Dog` and a structure that looks like a function too. You can see the parallels with function definitions which also have a name. The difference is that with functions we use the `def` keyword to define them, but with object definitions, we use the keyword "*class*".

Before we dive in and discuss what a `class` is, compared to an object, again have a look at some real but simple code that brings these abstract ideas to life. You can see the first line creates a variable called `sizzles`, and it seems to come from what looks like a function call. In fact that `Dog()` is a special function that creates an instance of the `Dog` class. We can see now how to create things from class definitions. These things are called *objects*. We have created an object called `sizzles` from the `Dog` class definition – we can

created an object called *sizzles* from the `Dog` class definition — we can consider this object to be a dog!

The next line calls the `bark()` function on the *sizzles* object. This is half familiar because we have seen functions already. What's less familiar is that we're calling the `bark()` function as if it was a part of the *sizzles* object. That's because `bark()` is a function that all objects created from the `Dog` class have. It's there to see in the definition of the `Dog` class. Let's say all that in simple terms. We created *sizzles*, a kind of `Dog`. That *sizzles* is an object, created in the shape of a `Dog` class. Objects are instances of a class. The following shows what we have done so far, and also confirms that ```sizzles.bark()``` does indeed output a “woof!”

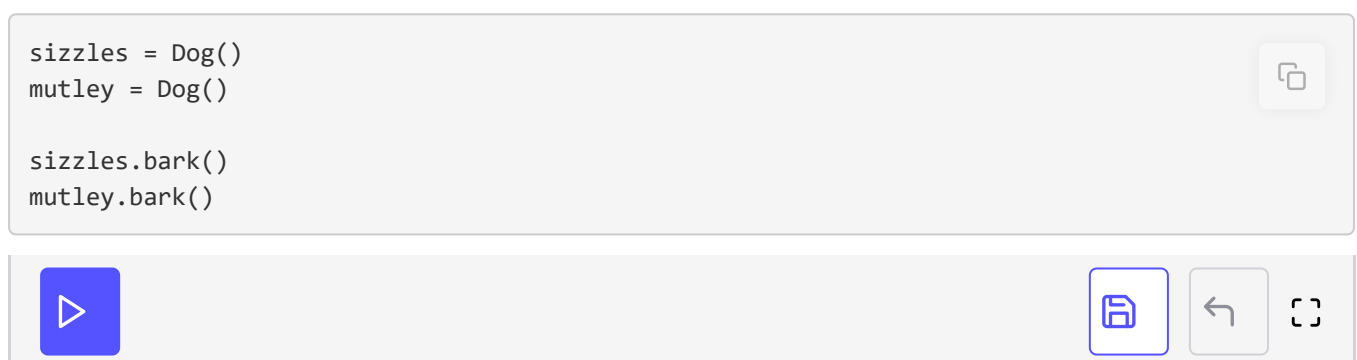
```
1 sizzles = Dog()
2 sizzles.bark()
```



You might have spotted the `self` in the definition of the function as `bark(self)`. This may seem odd, and it does to me. As much as I like Python, I don't think it is perfect. The reason that `self` is there is so that when Python creates the function, it assigns it to the right object. I think this should be obvious because that `bark()` is inside the class definition so Python should know which objects to connect it to, but that's just my opinion. Let's see objects and classes being used more usefully. Have a look at the following code. Let's run it and see what happens.

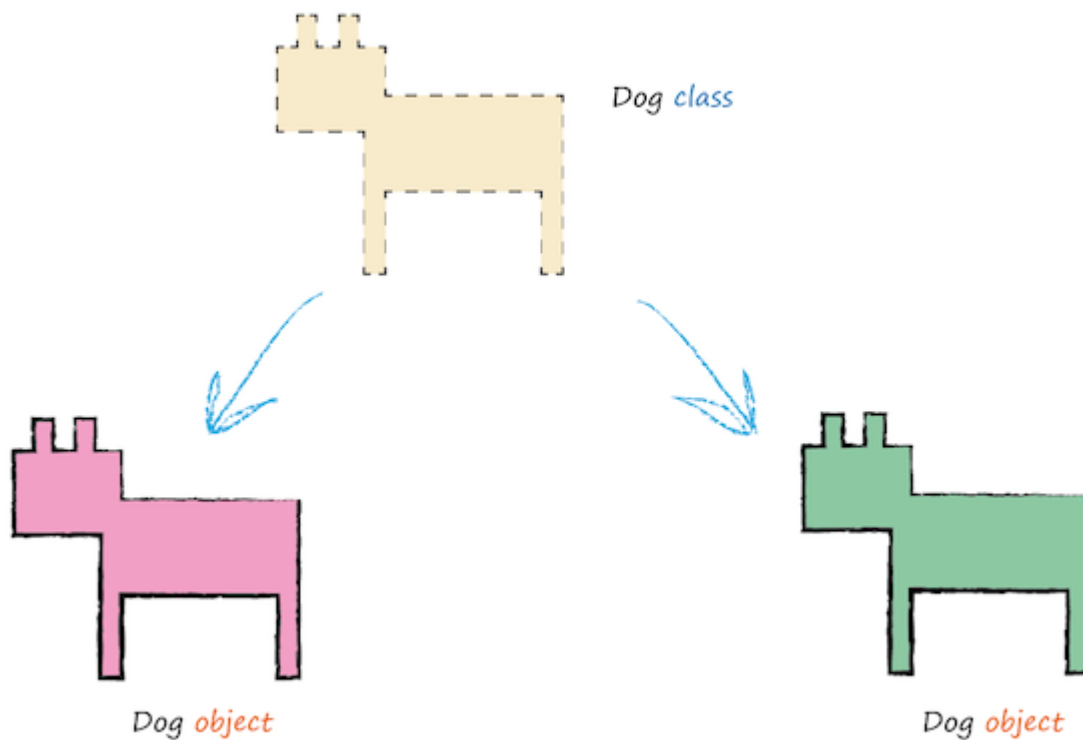
```
sizzles = Dog()
mutley = Dog()

sizzles.bark()
mutley.bark()
```



This is interesting! We are creating two objects called *sizzles* and *mutley*. The important thing to realize is that they are both created from the same `Dog()` class definition. This is powerful! We define what the objects should look like and how they should behave, and then we create real instances of them. That

is the difference between classes and objects, one is a definition and one is a real instance of that definition. A class is a cake recipe in a book, an object is a cake made with that recipe. The following shows visually how objects are made from a class recipe.



What use are these objects made from a class? Why go to the trouble? It would have been simpler to just print the word “woof!” without all that extra code. Well, you can see how it might be useful to have many objects all of the same kind, all made from the same template. It saves having to create each one separately in full. But the real benefit comes from objects having data and functions all wrapped up neatly inside them. The benefit is to us, human coders. It helps us more easily understand more complicated problems if bits of code are organized around objects to which they naturally belong. Dogs bark. Buttons click. Speakers emit sound. Printers print or complain they’re out of paper. In many computer systems, buttons, speakers, and printers are indeed represented as objects, whose functions you invoke.