Methods

What are methods and what is their use? This lesson will help you learn all about it.

You will sometimes see these object functions called *methods*. We've already done this in the previous lesson; we added a `bark()` function to the Dog class and both the sizzles and *mutley* objects made from this class have a `bark()` method. You saw them both bark in the example!

Neural networks take some input, do some calculations, and produce an output. We also know they can be trained. You can see that these actions, training and producing an answer, are natural functions of a neural network. That is functions of a neural network object. You will also remember that neural networks have data inside them that naturally belongs there — the link weights. That's why we will build our neural network as an object. For completeness, let's see how we add data variables to a class, and some methods to view and change this data. Have a look at the following fresh class definition of a Dog class. There are a few new things going on here so we'll look at them one at a time.

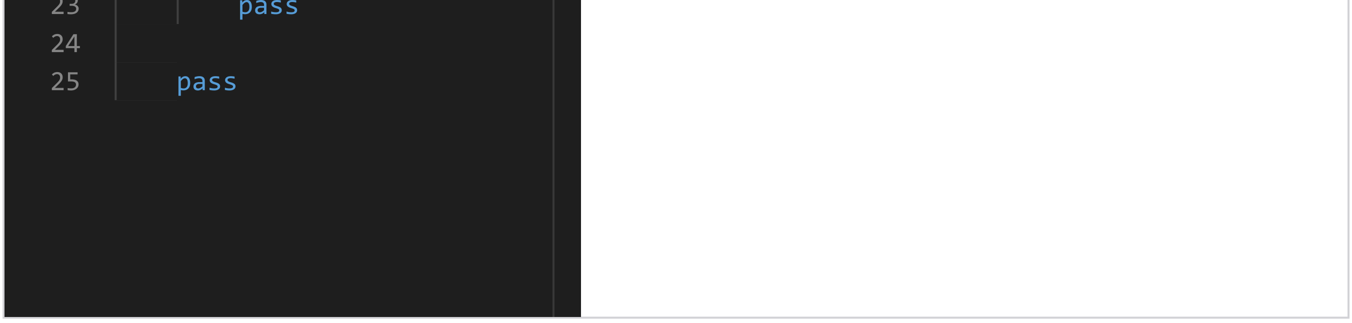
```
# class for a dog object
class Dog:

    # initialisation method wi
    def __init__(self, petname
        self.name = petname;
        self.temperature = tem

    # get status
    def status(self):
        print('dog name is ',
        print('dog temperature
        pass

    # set temperature
    def setTemperature(self,te
        self.temperature = tem
        pass

    # dogs can bark()
    def bark(self):
        print('woof!')
```



```
            pass

        pass
```

The first thing to notice is we have added three new functions to this Dog class. We already had the `bark()` function, now have new ones called `__init__()`, `status()` and `setTemperature()`. Adding new functions is easy enough to understand. We could have added one called `sneeze()` to go with `bark()` if we wanted.

But these new functions seem to have variable names inside the function names. That *setTemperature* is actually `setTemperature(self, temp)`. The funny named `__init__` is actually `__init__(self, petname, temp)`. What are these extra bits inside the brackets? They are the variables the function expects when it is called — called *parameters*. Remember that averaging function `avg(x,y)` we saw earlier? The definition of *avg()* made clear it expected 2 numbers. So the `__init__()` function needs a *petname* and a *temp*, and the `setTemperature()` function needs just a *temp*.

Now let's look inside these new functions. Let's look at that oddly named `__init__()` first. Why is it given such a weird name? The name is special, and Python will call a function called `__init__()` when an object is first created. That's really handy for doing any work preparing an object before we actually use it. So what do we do in this magic initialization function? We seem only to create two new variables, called `self.name` and `self.temperature`. You can see their values from the variables *petname*, and *temp* passed to the function. The `self.` part means that the variables are part of this object itself — it's own "self". That is, they belong only to this object, and are independent of another Dog object or general variables in Python. We don't want to confuse this dog's name with that of another dog! If this all sounds complicated, don't worry, it will be easy to see when we actually run a real example. Next is the `status()` function, which is really simple. It doesn't take any parameters, and simply prints out the Dog object's name and temperature variables.

Finally the `setTemperature()` function does take a parameter. You can see that it sets the `self.temperature` to the parameter temp supplied when this function is called. This means you can change the object's temperature at any time after you created the object. You can do this as many times as you like.

We've avoided talking about why all these functions, including that `bark()` function, have a "self" as the first parameter. It is a peculiarity of Python, which I find a bit irksome, but that's the way Python has evolved. What it does is make clear to Python that the function you're about to define belongs to the object, referred to as "self". You would have thought it was obvious because we're writing the function inside the class. You won't be surprised that this has caused debates amongst even expert Python programmers, so you're in good company in being puzzled. Let's see all this in action to make it really clear. The following shows the new Dog class defined with these new functions, and a new Dog object called *lassie* being created with parameters, setting out its name as "lassie" and its initial temperature as 37.

```
lassie = Dog('Lassie', 37)
lassie.status()
```

You can see how calling the `status()` function on this *lassie* Dog object prints out the dog's name and its current temperature. That temperature hasn't changed since *lassie* was created. Let's now change the temperature and check if it really has changed inside the object by asking for another update:

```
lassie.setTemperature(40)
lassie.status()
```

You can see that calling `setTemperature(40)` on the lassie object did indeed change the object's internal record of the temperature. We should be really pleased because we've learned quite a lot about objects, which some consider an advanced topic, and it wasn't that hard at all! We've learned enough Python to begin making our neural network.