## 20.13. A Tamagotchi Game

There are also a lot of interesting ways to put user-defined classes to use that don't involve data from the internet. Let's pull all these mechanics together in a slightly more interesting way than we got with the Point class. Remember Tamagotchis, the little electronic pets? As time passed, they would get hungry or bored. You had to clean up after them or they would get sick. And you did it all with a few buttons on the device.

We are going to make a simplified, text-based version of that. In your problem set and in the chapter on Inheritance <chap\_inheritance> we will extend this further.

First, let's start with a class Pet. Each instance of the class will be one electronic pet for the user to take care of. Each instance will have a current state, consisting of three instance variables:

- hunger, an integer
- · boredom, an integer
- · sounds, a list of strings, each a word that the pet has been taught to say

In the \_\_init\_\_ method, hunger and boredom are initialized to random values between 0 and the threshold for being hungry or bored. The \_sounds instance variable is initialized to be a copy of the class variable with the same name. The reason we make a copy of the list is that we will perform destructive operations (appending new sounds to the list). If we didn't make a copy, then those destructive operations would affect the list that the class variable points to, and thus teaching a sound to any of the pets would teach it to all instances of the class!

There is a clock\_tick method which just increments the boredom and hunger instance variables, simulating the idea that as time passes, the pet gets more bored and hungry.

The <u>\_str\_</u> method produces a string representation of the pet's current state, notably whether it is bored or hungry or whether it is happy. It's bored if the boredom instance variable is larger than the threshold, which is set as a class variable.

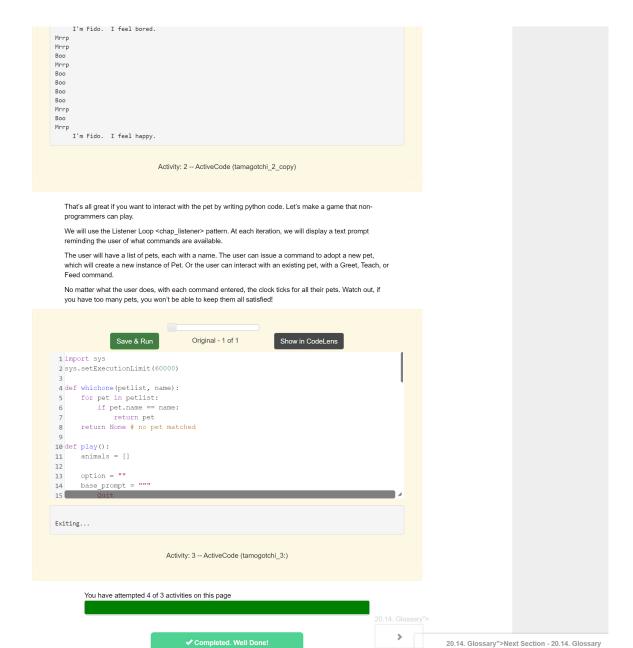
To relieve boredom, the pet owner can either teach the pet a new word, using the teach() method, or interact with the pet, using the hi() method. In response to teach(), the pet adds the new word to its list of words. In response to the hi() method, it prints out one of the words it knows, randomly picking one from its list of known words. Both hi() and teach() cause an invocation of the reduce\_boredom() method. It decrements the boredom state by an amount that it reads from the class variable

To relieve hunger, we call the feed() method.

```
Save & Run
                                    Original - 1 of 1
                                                          Show in CodeLens
37
       def teach(self, word):
           self.sounds.append(word)
38
           self.reduce_boredom()
40
      def feed(self):
41
          self.reduce hunger()
42
43
       def reduce_hunger(self):
45
           self.hunger = max(0, self.hunger - self.hunger decrement)
46
      def reduce_boredom(self):
47
          self.boredom = max(0, self.boredom - self.boredom_decrement)
49
                             Activity: 1 -- ActiveCode (tamagotchi 1)
```

Let's try making a pet and playing with it a little. Add some of your own commands, too, and keep printing p1 to see what the effects are. If you want to directly inspect the state, try printing p1.boredom or p1.hunger.

```
Original - 1 of 1
                                                                 Show in CodeLens
 1 pl = Pet("Fido")
 2 print(p1)
 3 for i in range(10):
     p1.clock_tick()
       print(p1)
 6 pl.feed()
 7 pl.hi()
 8 pl.teach("Boo")
 9 for i in range(10):
      p1.hi()
11 print (p1)
12
     I'm Fido. I feel happy
    I'm Fido. I feel happy.
I'm Fido. I feel bored.
     I'm Fido.
                I feel bored
     I'm Fido.
                I feel bored.
     I'm Fido. I feel bored.
     I'm Fido. I feel bored
     I'm Fido. I feel bored.
```



esting classes">

| Back to top

© Copyright 2017 bradleymiller. Created using Runestone 4.1.17.