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 (https://en.wikipedia.org/wiki/Tamagotchi), 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

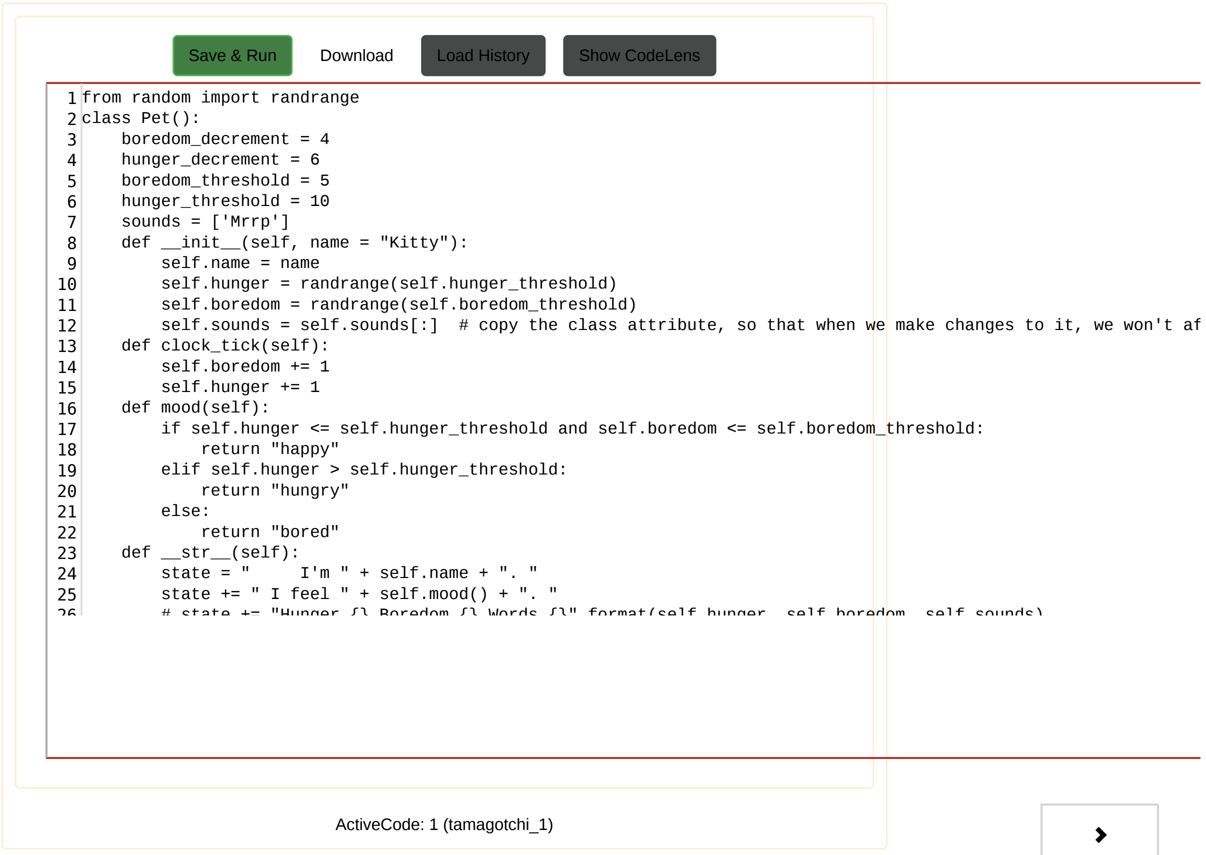
In the `__init__` method, hunger and boredom are initialized to random values between 0 and the threshold for being hungry or bored. The `sounds` instance variable is initialized to be a copy of the class variable with the same name. The reason we make a copy of the list is that we will perform destructive 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 `__str__` 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 hunger_decrement. The boredom state can never go below 0.

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



```
from random import randrange
class Pet():
    boredom_decrement = 4
    hunger_decrement = 6
    boredom_threshold = 5
    hunger_threshold = 10
    sounds = ['Mrrp']
    def __init__(self, name = "Kitty"):
        self.name = name
        self.hunger = randrange(self.hunger_threshold)
        self.boredom = randrange(self.boredom_threshold)
        self.sounds = self.sounds[:]  # copy the class attribute, so that when we make changes to it, we won't af
    def clock_tick(self):
        self.boredom += 1
        self.hunger += 1
    def mood(self):
        if self.hunger <= self.hunger_threshold and self.boredom <= self.boredom_threshold:
            return "happy"
        elif self.hunger > self.hunger_threshold:
            return "hungry"
        else:
            return "bored"
    def __str__(self):
        state = "     I'm " + self.name + ". "
        state += " I feel " + self.mood() + ". "
        # state += "Hunger {} Boredom {} Words {}".format(self.hunger, self.boredom, self.sounds)
```

ActiveCode: 1 (tamagotchi_1)

gClasses.html) (TestingClasses.html)

(Glossary.html) Next

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.

Save & Run Download Load History Show CodeLens

```
p1 = Pet("Fido")
print(p1)
for i in range(10):
    p1.clock_tick()
    print(p1)
p1.feed()
p1.hi()
p1.teach("Boo")
for i in range(10):
    p1.hi()
print(p1)

```

ActiveCode: 2 (tamagotchi_2_copy)

That's all great if you want to interact with the pet by writing python code. Let's make a game that non-programmers can play.

We will use the Listener Loop <chap_listener> pattern. At each iteration, we will display a text prompt reminding the user of what commands are available.

The user will have a list of pets, each with a name. The user can issue a command to adopt a new pet, which will create a new instance of Pet. Or the user can interact with an existing pet, with a Greet, Teach, or Feed command.

No matter what the user does, with each command entered, the clock ticks for all their pets. Watch out, if you have too many pets, you won't be able to keep them all satisfied!

Save & Run Download Load History Show CodeLens

```

    option = ""
    base_prompt = """
        Quit
        Adopt <petname_with_no_spaces_please>
        Greet <petname>
        Teach <petname> <word>
        Feed <petname>

        Choice: """
    feedback = ""
    while True:
        action = input(feedback + "\n" + base_prompt)
        feedback = ""
        words = action.split()
```

ActiveCode: 3 (tamogotchi_3:)

Mark as completed

username: sanjaysheel1997@gmail.com | Back to top

gClasses.html) (TestingClasses.html)

(Glossary.html) **Next**