CSc 110, Spring 2017

Lecture 38: Critters

Adapted from slides by Marty Stepp and Stuart Reges
Calling overridden methods

• Subclasses can call overridden methods with `super`
  
  ```python
  super(ClassName, self).method(parameters)
  ```

• Example:
  
  ```python
class Rabbit(Critter):
    def __init__(self):
      super(Rabbit, self).__init__()
      self.__moves = 0
      self.__hungry = False
  ```

What class is `Rabbit` inheriting from? ____________________

What method did `Rabbit` override above? ________________

What code creates an `instance` of the class `Rabbit`? ________________

What code would cause the `__str__` method of a class to be called? ________________
CSc 110 Critters

- Ant
- Bird
- Hippo
- Vulture
- WildCat (creative)

**behavior:**
- `eat` eating food
- `fight` animal fighting
- `get_color` color to display
- `get_move` movement
- `__str__` a single character to display
Inherit from the Critter class

• **Syntax:** class *name*(Critter):

class NewAnimal(Critter):
    def eat():
        # returns True or False
    def fight(opponent):
        # ROAR, POUNCE, SCRATCH
    def get_color():
        # returns a string for the color, e.g., "blue"
    def get_move():
        # returns NORTH, SOUTH, EAST, WEST, CENTER
    def __str__()
How the simulator works

• "Go" → loop:
  • move each animal (*get_move*)
  • if they collide, *fight*
  • if they find food, *eat*

• The simulator keeps score based on:
  • How many animals of that kind are still alive
  • How much food they have eaten
  • How many other animals they have beaten in a fight

• Simulator is in control!
  • *get_move* is **one move** at a time
    • *(no loops)*
  • Keep state (attributes)
    • to remember for future moves
Development Strategy

• Simulator helps you debug
  • smaller width/height
  • fewer animals
  • "Tick" instead of "Go"

• Write your own main
  • call your animal's methods and print what they return
class Critter():
    def eat(self):
        return False
    def fight(self, opponent):
        return ATTACK_FORFEIT
    def get_color(self):
        return "grey"
    def get_move(self):
        return DIRECTION_CENTER
    def __str__(self):
        return "?"
The Critter class constants

# Constants for attacks, directions
ATTACK_POUNCE = 0
ATTACK_ROAR = 1
ATTACK_SCRATCH = 2
ATTACK_FORFEIT = 3
DIRECTION_NORTH = 0
DIRECTION_SOUTH = 1
DIRECTION_EAST = 2
DIRECTION_WEST = 3
DIRECTION_CENTER = 4
Critter exercise: *Cougar*

- Write a critter class *Cougar*:

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td></td>
</tr>
<tr>
<td>eat</td>
<td>Always eats.</td>
</tr>
<tr>
<td>fight</td>
<td>Always pounces.</td>
</tr>
<tr>
<td>get_color</td>
<td>Blue if the <em>Cougar</em> has never fought; red if he has.</td>
</tr>
<tr>
<td>get_move</td>
<td>Walks west until he finds food; then walks east until he finds food; then goes west and repeats.</td>
</tr>
<tr>
<td><strong>str</strong></td>
<td>&quot;C&quot;</td>
</tr>
</tbody>
</table>
Critter exercise: Cougar

• We need to know two things about its state:
  • Has it ever fought?
  • How much food it has eaten? Needed in order to return the correct direction. (West/Eat/East/Eat/West/Eat/East, and so on)
• Two instance variables
  
  fought (of type bool)
  eaten (of type int)

• Method eat: increment eaten every time eat is called
• Method get_move: Walks west until he finds food; then walks east until he finds food; then goes west until west and repeats.
  
  if eaten is even, walk west else walk east
The Cougar class
rom Critter import *

class Cougar(Critter):
    # returns a Cougar
    def __init__(self):
        super(Cougar,self).__init__()  # call the superclass constructor
        self.__fought = False
        self.__eaten = 0

    # returns "C" as a representation of the cougar
    def __str__(self):
        return "C"

    # returns that the critter does want to eat
    def eat(self):
        self.__eaten += 1
        return True
The Cougar class- cont.

# returns the pounce attack
def fight(self, opponent):
    self.__fought = True
    return ATTACK_POUNCE

# returns west until the critter eats, returns east until it
# eats again and then repeats
def get_move(self):
    if(self.__eaten % 2 == 0):
        return DIRECTION_WEST
    else:
        return DIRECTION_EAST

# returns blue if the critter has never fought and red if it has
def get_color(self):
    if(not self.__fought):
        return "blue"
    else:
        return "red"
Debugging: Cougar

- Start small. Run the Cougar class.
  - In idle, create a Cougar object
  - Call the methods to verify the behavior
Debugging Cougar

```python
>>> c = Cougar()
'blue'
>>> c.get_move()
3
>>> c.eat()
True
>>> c.get_move()
2
>>> c.fight()
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    c.fight()
TypeError: fight() missing 1 required positional argument: 'opponent'
```
Debugging: Cougar

• Add Stone and Cougar to the list of methods Critters.py
• Use a small grid size, few animals
• Go tick by tick
• Simulator actions on each tick for each animal:
  • Move the animal (call get_move) in a random order
  • If moved to occupied square, call both animals fight methods
  • If moved onto food, call the animal's eat method.

• What the scores mean:
  • How many animals of the class are alive
  • How much food they have eaten
  • How many other animals they have destroyed in a fight
Ideas for state

• You must not only have the right state, but update that state properly when relevant actions occur.

• Counting is helpful:
  • How many total moves has this animal made?
  • How many times has it eaten? Fought?

• Remembering recent actions in attributes is helpful:
  • Which direction did the animal move last?
    • How many times has it moved that way?
  • Did the animal eat the last time it was asked?
  • How many steps has the animal taken since last eating?
  • How many fights has the animal been in since last eating?
Critter exercise: Aardvark

• Write a critter class `Aardvark`:

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code></td>
<td></td>
</tr>
<tr>
<td><code>eat</code></td>
<td>Eats 3 pieces of food and then stops</td>
</tr>
<tr>
<td><code>fight</code></td>
<td>randomly chooses between pouncing and roaring</td>
</tr>
<tr>
<td><code>get_color</code></td>
<td>pink if hungry and red if full</td>
</tr>
<tr>
<td><code>get_move</code></td>
<td>walks up two and then down two</td>
</tr>
<tr>
<td><code>__str__</code></td>
<td>&quot;a&quot; if hungry &quot;A&quot; otherwise</td>
</tr>
</tbody>
</table>
Critter exercise: Aardvark

• We need to know two things about its state:
  • How much food has it eaten?
  • How many moves has it taken?
• Instance variables:
  
  \[
  \begin{align*}
  &\text{eaten (of type int)} \\
  &\text{moves (of type int)}
  \end{align*}
  \]

• Method eat: increment eaten every time eat is called, return False after 3
• Method get_move: Walks up two and then down two
  
  \[
  \begin{array}{cccccccccccc}
  1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 & 1 & 2 & 3 & 4
  \end{array}
  \]
  \[\rightarrow\text{ use logic as in Rabbit}\]