CSc 110, Spring 2017

Lecture 37: Critters

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

• Subclasses can call overridden methods with `super`

  \[
  \text{super}(\text{ClassName, self}) \cdot \text{method} (\text{parameters})
  \]

• Example:

  ```python
  class LegalSecretary(Secretary):
      def get_salary(self):
          base_salary = super(LegalSecretary, self).get_salary()
          return base_salary + 5000.0
  ...
  ```

Name the superclass of `LegalSecretary` _______________________

What method did `LegalSecretary` override? _________________

What code creates an `instance` of the class `LegalSecretary`? _________________________
Inheritance and constructors

• Imagine that we want to give employees more vacation days the longer they've been with the company.
  • For each year worked, we'll award 2 additional vacation days.

• When an Employee object is constructed, we'll pass in the number of years the person has been with the company.

• This will require us to modify our Employee class and add some new state and behavior.

• Exercise: Make necessary modifications to the Employee class.
Modified Employee class

class Employee:
    def __init__(self, initial_years):
        self.__years = initial_years

    def get_hours(self):
        return 40

    def get_salary(self):
        return 40000.0

    def get_vacation_days(self):
        return 10 + 2 * self.__years

    def get_vacation_form(self):
        return "yellow"
Problem with constructors

• Now that we've added the constructor to the Employee class, an error is produced:

```python
TypeError: __init__() missing 1 required positional argument: 'initial_years'
```

• Short explanation: Once we write an `__init__(self, p1, … pn)` that requires parameters in the superclass, we must now write initialization methods for our employee subclasses as well.

• Exception: If the default behavior of the superclass is acceptable for all subclasses, you simply modify the class constructor expression.
Modified Marketer class

# A class to represent marketers.
class Marketer(Employee):
    def __init__(self, years):
        super(Marketer, self).__init__(years)

def advertise(self):
    print("Act now while supplies last!")

def get_salary():
    return super(Marketer, self).get_salary() + 10000.0

• Exercise: Modify the Secretary subclass.
  • Secretaries' years of employment are not tracked.
  • They do not earn extra vacation for years worked.
Modified Secretary class

# A class to represent secretaries.
class Secretary(Employee):
    def __init__(self):
        super(Secretary, self).__init__(0)

    def take_dictation(self, text):
        print("Taking dictation of text: " + text)

• Since Secretary doesn't require any parameters to its constructor, LegalSecretary does not require a constructor.
  • Its default constructor calls the Secretary constructor.
Inheritance and attributes

• Try to give lawyers $5000 for each year at the company:

```python
class Lawyer(Employee):
    ...
    def get_salary(self):
        return super(Lawyer, self).get_salary() + 5000 * self.__years
    ...
```

• Does not work; the error is the following:

```python
AttributeError: 'Lawyer' object has no attribute '_Lawyer__years'
```

• Private attributes cannot be directly accessed from subclasses.
  • One reason: So that subclassing can't break encapsulation.
  • How can we get around this limitation?
Add an accessor for any attribute needed by the subclass.

class Employee:
    def __init__(self, initial_years):
        self.__years = initial_years

    def get_years(self):
        return self.__years

... 

class Lawyer(Employee):
    def __init__(self, years):
        super(Lawyer, self).__init__(years)

    def get_salary(self):
        return super(Lawyer, self).get_salary() + 5000 * self.get_years()
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__ letter to display
Inherit from the Critter class

• Syntax: `class name(Critter):`

```python
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 \(\text{get\_move}\)
  • 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!
  • \(\text{get\_move}\) is one move at a time
    • \(\text{no loops}\)
  • 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
The Critter class

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><code>__init__</code></td>
<td></td>
</tr>
<tr>
<td><code>eat</code></td>
<td>Always eats.</td>
</tr>
<tr>
<td><code>fight</code></td>
<td>Always pounces.</td>
</tr>
<tr>
<td><code>get_color</code></td>
<td>Blue if the Cougar has never fought; red if he has.</td>
</tr>
<tr>
<td><code>get_move</code></td>
<td>Walks west until he finds food; then walks east until he finds food; then goes west and repeats.</td>
</tr>
<tr>
<td><code>__str__</code></td>
<td>&quot;C&quot;</td>
</tr>
</tbody>
</table>
Critter exercise: *Cougar*

- We need to know two things about its state:
  - If it has ever fought
  - How much food it has eaten in order to return the correct direction (West/Eat/East/Eat/West/Eat/East, and so on)

<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>Always eats.</td>
</tr>
<tr>
<td><code>fight</code></td>
<td>Always pounces.</td>
</tr>
<tr>
<td><code>get_color</code></td>
<td>Blue if the <em>Cougar</em> has never fought; red if he has.</td>
</tr>
<tr>
<td><code>get_move</code></td>
<td>Walks west until he finds food; then walks east until he finds food; then goes west and repeats.</td>
</tr>
<tr>
<td><code>__str__</code></td>
<td>&quot;C&quot;</td>
</tr>
</tbody>
</table>
from Critter import *

class Cougar(Critter):
    # returns a Cougar
    def __init__(self):
        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"
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: Anteater

• Write a critter class Anteater:

<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>Eats 3 pieces of food and then stops</td>
</tr>
<tr>
<td>fight</td>
<td>randomly chooses between pouncing and roaring</td>
</tr>
<tr>
<td>get_color</td>
<td>pink if hungry and red if full</td>
</tr>
<tr>
<td>get_move</td>
<td>walks up two and then down two</td>
</tr>
<tr>
<td><strong>str</strong></td>
<td>&quot;a&quot; if hungry &quot;A&quot; otherwise</td>
</tr>
</tbody>
</table>