# Cheat Sheet

|  |
| --- |
| Class Employee:  def \_\_init\_\_(self, years):  self.\_\_years = years    def get\_hours(self):  return 40    def get\_salary(self):  return 50000    def get\_vacation\_days(self):  return 10 + 2 \* self.\_\_years  def get\_years(self):  return self.\_\_years  class Lawyer (**Employee**):  def \_\_init\_\_(self, years):  **super(Lawyer, self).\_\_init\_\_()**  def get\_salary():  return  **super(Lawyer, self).get\_salary()**  + 5000.0 \* get\_years();    def get\_vacation\_days():  return  **super(Lawyer,**  **self).get\_vacation\_days()** + 5 |

### Inheritance

class name(superclass):

* **inheritance**: Forming a new class based on an existing class.
* **extend**: To inherit from another class.
* **superclass**: The "parent" class; the class being extended.
* **subclass**: The "child" class; the class extending another.
* **override**: To replace a superclass method with a new one.

### super keyword

#### Calling an overridden method from the superclass:

super(**classname**, self).method(parameters)

### Critter classes

class name (Critter):

fields

constructor

def eat(self):

statement(s) that return True (eat) or False (don't eat)

def fight(self, opponent):

statement(s) that return either ROAR, POUNCE, or SCRATCH

def get\_color():

statement(s) that return a Color

def get\_move():

statement(s) that return either NORTH, SOUTH, EAST, WEST, or CENTER

def \_\_str\_\_():

statement(s) that return a String

Your critter classes inherit several other methods. You can ignore them for the main animals Ant, Bird, Hippo, and Vulture. But for a creative WildCat, you may want to use them. Here are some of the methods:

def getx() # returns your Critter's x coordinate

def gety() # returns your Critter's y coordinate

def get\_width() # returns the width of the board

def get\_height() # returns the height of the board

def get\_neighbor(dir) # returns the critter 1 square away in given direction

For example, to find out whether a critter's x-coordinate is at least 5, or to find out whether a Frog is in the square east of your critter, you could write code like the following in your get\_move method:

if (getx() >= 5): if (get\_neighbor(EAST) == "F"):

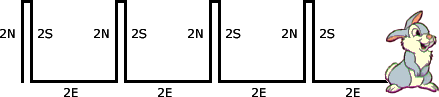
... ...

**Questions**

### Critter Classes

1. Define a Critter class named Rabbit with the following behavior:

|  |  |
| --- | --- |
| **constructor** | \_\_init\_\_() |
| **color** | dark gray (Color.DARK\_GRAY) |
| **eating behavior** | alternates between True and False (True, False, True, ...) |
| **fighting behavior** | if opponent is a Bird or Vulture, then scratch; otherwise, roar |
| **movement behavior** | hops in an L-shaped pattern:  first north twice, then south twice, then east twice, and repeat |
| **str** | "v" |



2. Define a Critter class named Frog with the following behavior:

|  |  |
| --- | --- |
| **constructor** | \_\_init\_\_(self, age)  The age passed will be between 1 and 9 inclusive. |
| **color** | green |
| **eating behavior** | never eats (this is the default eating behavior) |
| **fighting behavior** | always forfeits in a fight (this is the default fighting behavior) |
| **movement behavior** | moves east sometimes, or stays put (center), based on the frog's age:  If the frog is 1 year old, moves to the east every move.  E, E, E, E, E, E, E, E, E, E, ...  If the frog is 2 years old, moves to the east once every 2 moves.  C, E, C, E, C, E, C, E, C, E, ...  If the frog is 3 years old, moves to the east once every 3 moves.  C, C, E, C, C, E, C, C, E, C, ...  ...  If the frog is 9 years old, moves to the east once every 9 moves.  C, C, C, C, C, C, C, C, E, C, ... |
| **\_\_str\_\_** | "F" |



(continued on back page)

# Questions (continued)

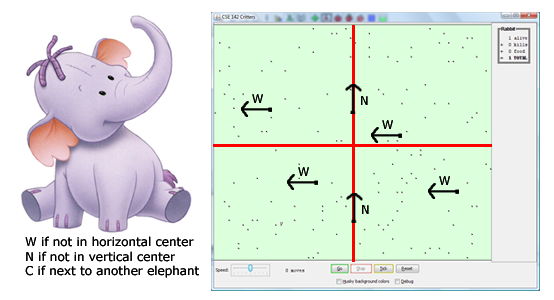
3. Define a Critter class named Elephant with the following behavior:

|  |  |
| --- | --- |
| **constructor** | \_\_init\_\_(self, attack)  (attack will be ROAR, POUNCE, or SCRATCH) |
| **color** | gray ("gray") |
| **eating behavior** | upon birth, decides either to always eat (True) or never eat (False) throughout its lifetime |
| **fighting behavior** | uses the attack that was passed to the constructor |
| **movement behavior** | prefers to move west if it is not in the horizontal center of the world;  otherwise, prefers north if it is not in the vertical center of the world;  but will not move onto a square occupied by another Elephant |
| **\_\_str\_\_** | "J" if this Elephant is hungry (if eat would return True); otherwise "j" |

The constructor accepts a parameter specifying which attack it should use when fight is called on that object. Also, when it is born, each Elephant object should make a random choice that it will either always eat (True) or never eat (False) during its lifetime. A given Elephant object should remember its choice and always return this same choice throughout its lifetime, no matter how many times eat is called on it.

Out of fear of predators, all Elephants try to move to the center of the simulation world to be together in a herd. They do this by first moving west until they reach the central x-coordinate on the board (half the board's width), then moving north until they reach the central y-coordinate on the board (half the board's height). Because mating leaves two critters vulnerable for several moves, Elephants do not want to mate; no Elephant will move onto a square occupied by another Elephant. If the Elephant's otherwise preferred move is occupied by another Elephant, it will instead choose to stay still (a move of CENTER).

To implement the movement behavior described previously, you will need to use some of the other inherited methods described on the Cheat Sheet and on page 4 of the assignment spec.



# Solutions

1.

# this solution wouldn't get full credit for commenting

class Rabbit(Critter):

def \_\_init\_\_(self):

self.\_\_moves = 0

self.\_\_hungry = False

def eat(self):

self.\_\_hungry = not self.\_\_hungry # reverse the boolean value

return self.\_\_hungry

def fight(self, opponent):

if (opponent == "^" or opponent == ">" or

opponent or "V" or opponent == "<"): # a Bird or Vulture

return SCRATCH

else:

return ROAR

def get\_color(self):

return "dark gray"

def get\_move(self): # "hops" north 2, south 2, east 2

self.\_\_moves += 1

if (self.\_\_moves > 6):

self.\_\_moves = 1 # start over

if (self.\_\_moves <= 2): # 1st or 2nd move

return NORTH

else if (\_\_moves <= 4): # 3rd or 4th

return SOUTH

else: # 5th or 6th

return EAST

def \_\_str\_\_():

return "V"

2.

# this solution wouldn't get full credit for commenting

class Frog(Critter):

def \_\_init\_\_(self, age):

self.\_\_age = age

self.\_\_count = 0

def get\_color():

return "green"

def get\_move():

\_\_self.\_\_count += 1

if (self.\_\_count >= age): # go EAST once every 'age' moves

\_\_self.\_\_count = 0

Return EAST

else:

return CENTER

def \_\_str\_\_():

return "F"

# Solutions (continued)

3.

# this solution wouldn't get full credit for commenting

# Elephants move west, then north, until they reach the world's center.

class Elephant(Critter)

def \_\_init\_\_(self, attack):

self.\_\_attack = attack

self.\_\_random\_num = randint(0, 1) # choose random value for all future eats

def eat(self):

# 50/50 chance based on above code

return (self.\_\_random\_num == 1)

def fight(self, opponent):

return self.\_\_attack

def get\_color():

return "gray"

def get\_move():

direction = CENTER

if (getx() != self.get\_width() // 2):

direction = WEST

else if (gety() != self.get\_height() // 2):

direction = NORTH

neighbor = get\_neighbor(direction) # halt if an Elephant is there

if (neighbor == self.\_\_str\_\_()):

direction = CENTER

return direction

def \_\_str\_\_():

if (random\_num == 1):

return "J"

else:

return "j"

Solutions from section #12

4.

Class BookData:

def \_\_init\_\_(self, author, title, rating):

self.\_\_author = author

self.\_\_title = title

self.\_\_rating = rating

def get\_title(self):  
 return self.\_\_title

def get\_author(self):

return self.\_\_author

def get\_rating(self):

return self.\_\_rating

def set\_rating(self, new\_rating):

self.\_\_rating = new\_rating

def main():  
book = BookData("Rowling", "Harry Potter", 5)

print(book.get\_title())

book.set\_rating(10)

print(book.get\_rating())

5.

class AdmissionsEntry:

def \_\_init\_\_(self, id, first, last, flagged, ratings):

self.\_\_id = id

self.\_\_first\_name = first

self.\_\_last\_name = last

self.\_\_flagged = flagged

self.\_\_ratings = ratings

def rate(self, rating):

self.\_\_ratings.append(rating)

def flag(self):

self.\_\_flagged = True

def get\_id(self):

return self.\_\_id

def get\_rating():

rating = 0

for rate in self.\_\_ratings:

rating += rate

return rating / len(self.\_\_ratings)

def main():

lines = open("data.txt").readlines()

candidates = []

avg\_rating = 0

for line in lines:

line = line.split(5)

entry = AdmissionsEntry(line[0], line[1], line[2], line[3], line[4])

avg\_rating += entry.get\_rating()

print("The average for all candidates: " + str(avg\_rating / len(candidates)))