Case Study
Bears, Fish, and Plants, Oh My!

Special Topic in Computer Science
Python Programming

Objectives

- To explore classes and objects further
- To design a large multiclass application
- To implement a graphical simulation using objects

What is Agent-Based Modeling (ABM) ?

- Simulation modeling technique where a system is modeled as a collection of agents and the relationships between them.

- Agents individually asses its situation in the environment and make decisions on the basis of a set of rules.

What Is a Simulation?

- A computer simulation is a computer program that is designed to model some specific aspects of a real situation or system.
- Computer simulations will often utilize random numbers as a way of introducing some realistic variability into the underlying model.
- The results can then be used to gain information about the way a real system behaves.
- The complexity of the computer simulation is dependent on the complexity of the underlying reality as well as the specific characteristics we are interested in representing.

What Is a Simulation? (cont’d.)

- One of the most common types of simulation is one in which a "real world" relationship is modeled, typically between two or more life-forms that must coexist and depend on one another in some fashion.
- These relationships, often referred to as predator-prey relationships, suggest that one life-form (the predator) preys on another (the prey) in order to survive.
- As the number of predators increases or decreases, it may in turn cause the number of prey to increase or decrease.
- Examples such as foxes and rabbits, big fish and little fish, and whales and plankton are common.
- In each of these cases one life-form consumes another and the existence of one depends on the other.

Rules of the Game

- In this chapter, we will construct a computer simulation that will graphically show a world that is inhabited by both predator and prey.
- In particular, we will model bears and fish.
- There will be rules that each must live by and as time progresses, individuals will live, breed, die, and move about.
- We will be able to observe the impact of initial conditions as well as the interaction of one life-form with another as the simulation plays out.

Rules of the Game (cont'd.)

- Our computer simulation models a world that contains two types of individual life-forms: bears and fish.
- We can think of the world as a two-dimensional "grid" with a fixed size for each dimension.
- Life-forms will only be able to live at specific locations within the grid.
- Each life-form will be described by a set of rules that governs how it lives.
- To start, a group of life-forms will be placed in the world at random locations.

Rules of the Game (cont'd.)

- The simulation will progress by allowing one of the life-forms to live for one time unit.
- During this unit of time, all other life-forms are in "suspended animation."
- This means that at any given time the life-form will be in one of two states: (1) alive or (2) suspended.
- The particular life-form will be chosen at random from the collection of all possible life-forms.
- Each time a life-form is in the alive state it must reevaluate its surroundings because it is likely that the rest of the world will have changed during the previous time units.
- It is in this way that the simulation takes on a sense of reality.

Rules of the Game (cont'd.)

- Fish will be allowed to breed, move, and die.
- Once a fish has been in the alive state twelve times, it may attempt to *breed*.
  - To do so, it will randomly pick an adjacent location.
  - If that location is empty, a new fish will appear.
  - If the location is occupied, the fish will have to wait until next time and try again.
- Regardless of whether a fish breeds, it will next try to *move*.
  - When a fish moves, it will randomly pick an adjacent location.
  - If that location is empty, the fish will move to this new location.
  - If the location is occupied, the fish will remain in its current location.

Rules of the Game (cont'd.)

- One additional environmental characteristic that will affect fish is overcrowding. If a fish discovers that there are two or more other fish living adjacent, then the fish will *die*.
- This means that even in the complete absence of bears, fish will self-regulate to some extent.
- In addition, in this version of the simulation fish never need to eat.
- Bears will breed, move, eat, and die.
- *Breeding* will take place in much the same fashion as described for fish. The only difference will be that bears need to be in the alive state eight times before they can start the breeding process.
- Bears will *move* in exactly the same manner as fish.

Rules of the Game (cont'd.)

- Bears will not be impacted by overcrowding but they will need to eat.
- In order to *eat*, a bear will need to determine whether there are fish living in an adjacent location.
  - If they are, then the bear will randomly pick one of the fish and "eat" it.
  - In order to consume the fish, the bear will move to the location currently occupied by the chosen fish.
  - If there are no adjacent fish , the bear will begin to starve.
- Any bear that has been in the alive state and starving ten times in a row will *die*.

Design

- The first step is to identify those parts of the problem that correspond to objects.
- One of the easiest ways to start the process of object identification is to consider the prominent **nouns** that appear in the description of the problem.
- In this case, words like *bear, fish,* and *world* seem like good choices.
- The nouns that you identify now will likely become Python classes when we implement the design.

Design (cont’d.)

- Next, we need to analyze each noun and come up with a list of things that it should know and a list of actions that it can perform.
- The list of things that an object should know will become instance variables.
- The list of things that an object can do will become methods.
- This process will be very helpful in deciding how objects will interact with one another.

Design (cont'd.)

- A world should know
  - The maximum X and Y dimensions.
  - A collection of all life-forms present.
  - A grid with the locations of each specific life-form.
- A world should be able to
  - Return the dimensions of the world.
  - Add a new life-form at a specific location.
  - Delete a life-form whereever it is.
  - Move a life-form from one world location to another.
  - Check a world location to see if it is empty.
  - Return the life-form at a specific location.
  - Allow a life-form to live for one time unit.
  - Draw itself.

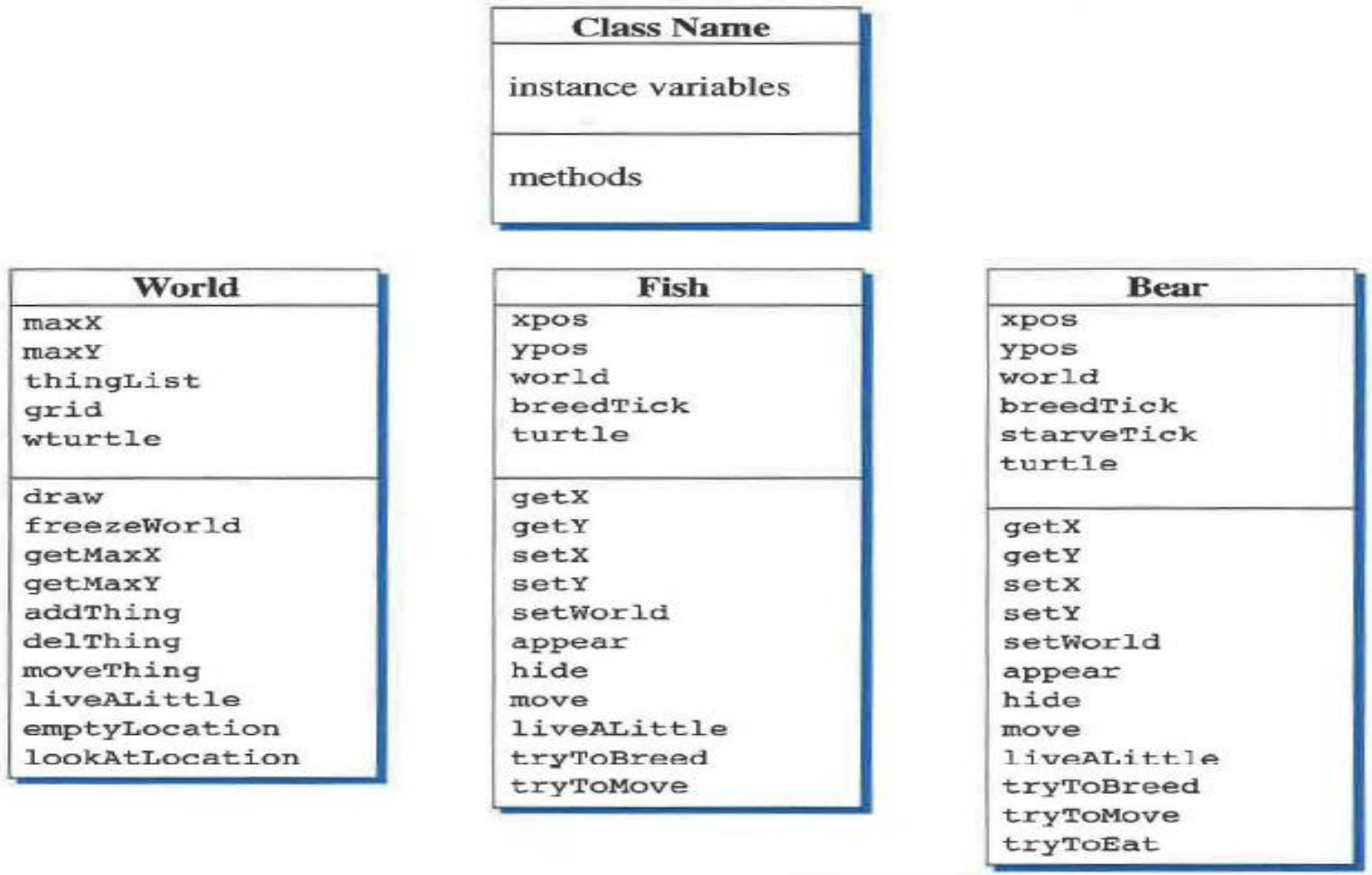
Design (cont'd.)

- A bear should know
  - The world it belongs to.
  - Its specific world location given as (x, y).
  - How long it has gone without food.
  - How long it has gone without breeding.
- A bear should be able to
  - Return its location, both the x and y value.
  - Set its location, both the x and y value.
  - Set the world that it belongs to.
  - Appear if it is newly born.
  - Hide if it dies.
  - Move from one location to another.
  - Live for one time unit (includes breeding, eating, moving, and dying).

Design (cont'd.)

- A fish should know
  - The world it belongs to.
  - Its specific location given as (x, y).
  - How long it has gone without breeding.
- A fish should be able to
  - Return its location, both the x and y value.
  - Set its location, both the x and y value.
  - Set the world that it belongs to.
  - Appear if it is newly born.
  - Hide if it dies.
  - Move from one location to another.
  - Live for one time unit (includes breeding, moving, and dying).

Design (cont'd.)



Listing instance variables and methods for each class

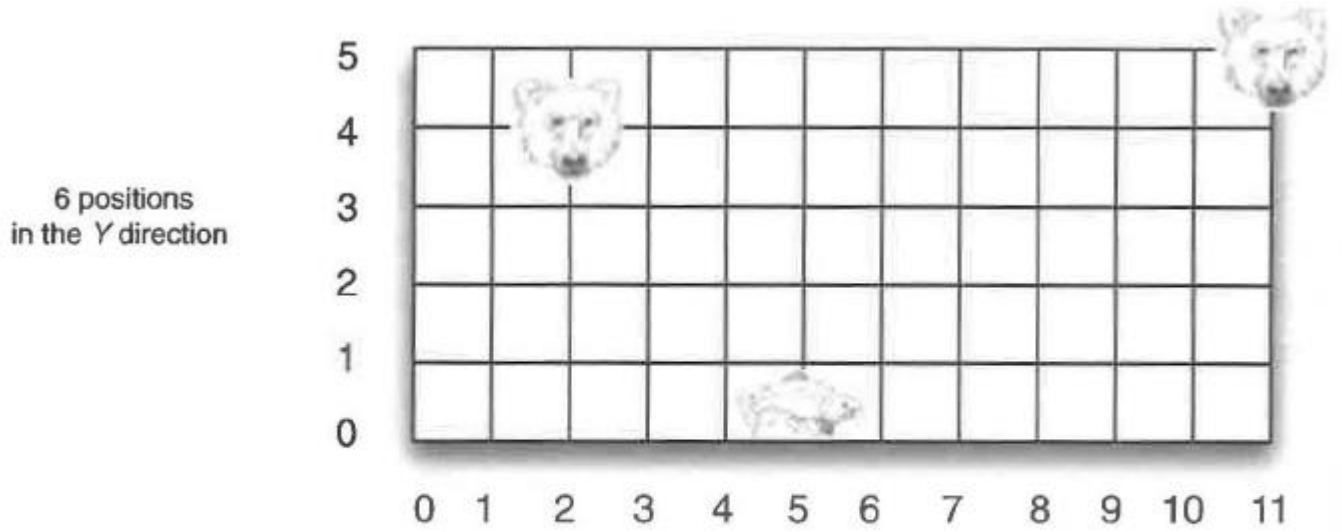
Implementation

- We will create our simulation by first implementing the instance variables and methods of the classes as described previously.
- Once we have the classes in place, we can complete the simulation by simply creating a world, adding life-forms to it, and then allowing the life-forms to live.

The World Class

- The World class will maintain dimensions as well as a list of the life-forms that are present.
- It will also have a grid to track the life-form positions.
  - A Turtle will be used to set the initial coordinates for the world and to draw the grid.
- The grid will maintain the exact two-dimensional position of each life-form.
- Each entry in the grid may contain a reference to a life-form.
  - For those grid positions that do not contain a life-form, we will use the None value.

The World Class (cont'd.)



A grid with three items

The World Class (cont'd.)

- One of the easiest ways to implement a collection in Python is to use a list. Since lists are one-dimensional, we need to represent the grid's two dimensional structure.
- The solution will be to implement the rows and columns of the grid using a "list of lists" structure.

```
g = [   [ None,None,None,None,None,FISH,None,None,None,None,None,None ] ,
        [ None,None,None,None,None,None,None,None,None,None,None,None ] ,
        [ None,None,None,None,None,None,None,None,None,None,None,None ] ,
        [ None,None,None,None,None,None,None,None,None,None,None,None ] ,
        [ None,None,BEAR,None,None,None,None,None,None,None,None,None ] ,
        [ None,None,None,None,None,None,None,None,None,None,None,BEAR ]    ]
```

A list of lists representation for the example grid

The World Class (cont'd.)

- Each item in the list g corresponds to a row in the grid.
- Within each row there are 12 items. To access a specific item simply requires another list index.
- If we want to access the item at any *(x, y)* location, the list of lists grid location will be given by g [y] [x].
  - The first index will be the row or y value and the second index will be the column or x value.

The World Class (cont'd.)

- The constructor for the World class, creates five instance variables: (1) maxX, (2) maxY, (3) thingList, (4) wturtle, and (5) grid.
- Initially grid is an empty list.
- A nested iteration is used to create the list of lists implementation.
  - Each row is first created by repeatedly appending None to an empty list.
  - The entire list is then appended to the list of lists (self.grid).
- The remainder of the constructor modifies the coordinate system and adds two new shapes that will be used later as icons for fish and bears.
- The wturtle function is hidden so that the default triangle shape is no longer present.

The World Class (cont'd.)

```
class World:
    def __init__(self, mx, my):
        self.maxX = mx
        self.maxY = my
        self.thingList = []
        self.grid = []

        for arow in range(self.maxY):
            row = []
            for acol in range(self.maxX):
                row.append(None)
            self.grid.append(row)

        self.wturtle = turtle.Turtle()
        self.wscreen = turtle.Screen()
        self.wscreen.setworldcoordinates(0,0,self.maxX-1,self.maxY-1)
        self.wscreen.addshape("Bear.gif")
        self.wscreen.addshape("Fish.gif")
        self.wturtle.hideturtle()
```

The World Class (cont'd.)

- The draw method will use wturtle to draw the grid system assuming the maximum *x* and *y* dimensions.
  - It first draws the outer boundaries of the grid and then goes back and fills in the horizontal and vertical lines.
- The other World methods will allow us to access or change some aspect of the simulation world.
  - accessor methods getMaxX and getMaxY will return the maximum dimensions.
  - emptyLocation will return True or False depending on whether there is a life-form at that particular location in the grid.
  - lookAtLocation will return the value at a particular grid location.

The World Class (cont'd.)

```
def draw(self):
    self.wscreen.tracer(0)
    self.wturtle.forward(self.maxX-1)
    self.wturtle.left(90)
    self.wturtle.forward(self.maxY-1)
    self.wturtle.left(90)
    self.wturtle.forward(self.maxX-1)
    self.wturtle.left(90)
    self.wturtle.forward(self.maxY-1)
    self.wturtle.left(90)
    for i in range(self.maxY-1):
        self.wturtle.forward(self.maxX-1)
        self.wturtle.backward(self.maxX-1)
        self.wturtle.left(90)
        self.wturtle.forward(1)
        self.wturtle.right(90)
    self.wturtle.forward(1)
    self.wturtle.right(90)
    for i in range(self.maxX-2):
        self.wturtle.forward(self.maxY-1)
        self.wturtle.backward(self.maxY-1)
        self.wturtle.left(90)
        self.wturtle.forward(1)
        self.wturtle.right(90)
    self.wscreen.tracer(1)
```

```
def getMaxX(self):
    return self.maxX

def getMaxY(self):
    return self.maxY

def emptyLocation(self,x,y):
    if self.grid[y][x] == None:
        return True
    else:
        return False

def lookAtLocation(self,x,y):
   return self.grid[y][x]
```

The World Class (cont'd.)

- The freezeWorld method is used at the end of the simulation. It calls the exitOnClick method of wturtle to allow the graphics window to remain drawn after the simulation has ended. Clicking on the window will then exit the simulation.
- A new life-form is added to the world by addThing. It needs the life-form and the *(x, y)* position where the life-form should be placed. It adds the life-form to the grid and appends the life-form to the list of life-forms maintained by the world.
- Removing a life-form from the simulation world requires that it be taken off the grid and also removed from the life-form list. This is done in the delThing method. It sets the appropriate grid reference to None and uses the list method remove to delete the fish or bear from the list of life-forms maintained by the world.

The World Class (cont'd.)

```
def freezeWorld(self):
    self.wscreen.exitonclick()

def addThing(self, athing, x, y):
    athing.setX(x)
    athing.setY(y)
    self.grid[y][x] = athing
    athing.setWorld(self)
    self.thingList.append(athing)
    athing.appear()

def delThing(self,athing):
    athing.hide()
    self.grid[athing.getY()][athing.getX()] = None
    self.thingList.remove(athing)

def moveThing(self,oldx,oldy,newx,newy):
    self.grid[newy][newx] = self.grid[oldy][oldx]
    self.grid[oldy][oldx] = None
```

The World Class (cont'd.)

- The liveALittle method does the majority of the simulation work.
- As long as there are still life-forms remaining, the world will pick a random life-form from the list that it maintains.
- Once a random life-form has been chosen, it is allowed to live as was described earlier.
- Note that the world lets only one life-form live during every call to liveALittle.

```
def liveALittle(self):
    if self.thingList != [ ]:
       athing = random.randrange(len(self.thingList))
       randomthing = self.thingList[athing]
       randomthing.liveALittle()
```

The Fish Class

- In this simulation, fish are the prey. They will try to breed, move around, and inevitably may end up in the vicinity of a bear who will likely eat them for dinner.
- Fish need to know where they are and how long it has been since they have given birth.

The Fish Class (cont'd.)

- The instance variables of the fish constructor for the Fish class, self.xpos and self.Ypos, are initialized to (0,0) but will be set later when the fish is placed in the world.
- Each fish will have a reference back to the world that they live in. The reference is set to None.
- The next group of methods for the Fish class will be the simple accessor and mutator methods used by other objects as they interact with Fish objects.
- When a Fish is first created, it will be added to the world at a specific location and then told to appear. The appear method will take care of moving the underlying Turtle object and then showing the icon in the simulation display.
- Once the Fish object is in the simulation, it can be moved to a new location using the move method. This method not only moves the underlying Turtle but also changes the *x* and *y* positions of the Fish object.

The Fish Class (cont'd.)

```
class Fish:
    def __init__(self):
        self.turtle = turtle.Turtle()
        self.turtle.up()
        self.turtle.hideturtle()
        self.turtle.shape("Fish.gif")

        self.xpos = 0
        self.ypos = 0
        self.world = None

        self.breedTick = 0

    def setX(self,newx):
        self.xpos = newx

    def setY(self,newy):
        self.ypos = newy

    def getX(self):
        return self.xpos

    def getY(self):
        return self.ypos
```

```
    def setWorld(self,aworld):
        self.world = aworld

    def appear(self):
        self.turtle.goto(self.xpos, self.ypos)
        self.turtle.showturtle()

    def hide(self):
        self.turtle.hideturtle()

    def move(self,newx,newy):
        self.world.moveThing(self.xpos,self.ypos,newx,newy)
        self.xpos = newx
        self.ypos = newy
        self.turtle.goto(self.xpos, self.ypos)
```

The Fish Class (cont'd.)

- The IiveALittle method is responsible for what happens to a Fish each time it is allowed to live for a time unit.
- Fish can die if they are crowded by too many other fish. If that is not the case, then they will try to breed and then finally try to move. If a fish dies due to overcrowding, it will not attempt to breed or move.
- The first thing this method needs to do is count the number of Fish that are at adjacent locations in order to decide if overcrowding is occurring. This will require that we design a process that allows us to find all of the locations that are adjacent to a given location.
- For this task, we will use a list of offsets. Each offset is a tuple containing an amount to adjust the *x* coordinate and an amount to adjust the *y* coordinate.
- we can take the coordinates, reduce them to an offset tuple, and then simply create a list of these offset tuples. Now it is just a matter of iterating through the offsetList and adjusting the original *(x, y)* position by the appropriate tuple component.

The Fish Class (cont'd.)

$(x-1, y+1)$	$(x+0, y+1)$	$(x+1, y+1)$
$(x-1, y+0)$	(x, y)	$(x+1, y+0)$
$(x-1, y-1)$	$(x+0, y-1)$	$(x+1, y-1)$

Coordinates of the eight adjacent locations around a specific (x, y)

$(-1, +1)$	$(0, +1)$	$(+1, +1)$
$(-1, 0)$	(x, y)	$(+1, +0)$
$(-1, -1)$	$(0, -1)$	$(+1, -1)$

Coordinates of the eight adjacent locations around a specific (x, y)

The Fish Class (cont'd.)

```
def liveALittle(self):
    # creates the list of offset tuples
    offsetList = [(-1,1) ,(0,1) ,(1,1),
                  (-1,0)        ,(1,0),
                  (-1,-1),(0,-1),(1,-1)]
    adjfish = 0  # variable to count the number of neighboring fish
    for offset in offsetList:
        # new x and y values are computed from the current (x, y)
        # location and the current offset tuple.
        newx = self.xpos + offset[0]
        newy = self.ypos + offset[1]
        # if this new location is actually a legal location (not on a boundary)
        if 0 <= newx < self.world.getMaxX()  and
           0 <= newy < self.world.getMaxY():
            # check if an object is an instance of a particular class
            if (not self.world.emptyLocation(newx,newy)) and
                isinstance(self.world.lookAtLocation(newx,newy),Fish):
                adjfish = adjfish + 1
    # if there is overcrowding, delete the fish;
    # else, check whether the fish has been active for enough time units to try to breed
    if adjfish >= 2:
        self.world.delThing(self)
    else:
        self.breedTick = self.breedTick + 1
        if self.breedTick >= 12:
            self.tryToBreed()
        # move to a new location.
        self.tryToMove()
```

The Fish Class (cont'd.)

- If it is determined that the Fish can try to breed, the tryToBreed method is called.
- According to the rules, this method must first pick a random adjacent location.
- To do this, we can use the same offset list technique that we used earlier. In this case, instead of iterating through all of the offsets, we will simply pick one.
- In order to choose a random element of the list, we will first choose a random integer in the range of index values using the length of the offsetList as the upper bound.
- Once we compute this new *(x, y)* position, we must be sure that it is in the actual range of legal coordinates. If it is not, we must try again.

The Fish Class (cont'd.)

```
def tryToBreed(self):
    offsetList = [(-1,1) ,(0,1) ,(1,1),
                  (-1,0)        ,(1,0),
                  (-1,-1),(0,-1),(1,-1)]

    # continues to choose random offset pairs
    # until a legal result is obtained
    randomOffsetIndex = random.randrange(len(offsetList))
    randomOffset = offsetList[randomOffsetIndex]
    nextx = self.xpos + randomOffset[0]
    nexty = self.ypos + randomOffset[1]
    while not (0 <= nextx < self.world.getMaxX() and
               0 <= nexty < self.world.getMaxY() ):
        randomOffsetIndex = random.randrange(len(offsetList))
        randomOffset = offsetList[randomOffsetIndex]
        nextx = self.xpos + randomOffset[0]
        nexty = self.ypos + randomOffset[1]

    # checks the location to see if any life-forms are present.
    # If they are not, a new Fish is created and added to the world at that location.
    if self.world.emptyLocation(nextx,nexty):
       childThing = Fish()
       self.world.addThing(childThing,nextx,nexty)
       self.breedTick = 0
```

The Fish Class (cont'd.)

- The tryToMove method has much of the same functionality as tryToBreed. It must first pick a random adjacent location and check if it is empty. If it is, the fish is allowed to move using the move m

```
def tryToMove(self):
    offsetList = [(-1,1) ,(0,1) ,(1,1),
                  (-1,0)        ,(1,0),
                  (-1,-1),(0,-1),(1,-1)]
    randomOffsetIndex = random.randrange(len(offsetList))
    randomOffset = offsetList[randomOffsetIndex]
    nextx = self.xpos + randomOffset[0]
    nexty = self.ypos + randomOffset[1]
    while not(0 <= nextx < self.world.getMaxX() and
              0 <= nexty < self.world.getMaxY() ):
        randomOffsetIndex = random.randrange(len(offsetList))
        randomOffset = offsetList[randomOffsetIndex]
        nextx = self.xpos + randomOffset[0]
        nexty = self.ypos + randomOffset[1]

    if self.world.emptyLocation(nextx,nexty):
       self.move(nextx,nexty)
```

The Bear Class

- The Bear class is implemented in much the same way as the Fish class.
- We will focus on those methods that are either different or are new.
- The obvious place to start is with the liveALittle method and the constructor.
- As described earlier, bears will not be impacted by overcrowding. The first thing they will do is try to breed.
- The other important difference is that bears will eat fish if fish are available. Any bear that does not get enough food will starve and die.
- The constructor for the Bear class will need to include one more instance variable that will keep track of the number of time units since the bear's last meal.
- The liveALittle method allows a bear to try to eat a fish.

The Bear Class (cont'd.)

```
class Bear:
    def __init__(self):
        self.turtle = cTurtle.Turtle()
        self.turtle.up()
        self.turtle.hideturtle()
        self.turtle.shape("Bear.gif")

        self.xpos = 0
        self.ypos = 0
        self.world = None

        self.starveTick = 0
        self.breedTick = 0
```

```
def liveALittle(self):
    self.breedTick = self.breedTick + 1
    if self.breedTick >= 8:
        self.tryToBreed()

    self.tryToEat()

    if self.starveTick == 10:
        self.world.delThing(self)
    else:
        self.tryToMove()
```

The Bear Class (cont'd.)

- In tryToEat method, bears will check in the adjacent locations to see if there are any fish present.
- If so, the bear will pick one of the fish at random and eat it.
  - This means that the bear moves to the location currently occupied by the fish and the fish dies.
- If there are no fish present, the bear starves a bit more.
- It is important to note that in the liveALittle method for Bear, the tryToMove method is done regardless of whether the bear eats.
- This means that it is possible to have the bear move twice during a time unit, once to eat and then again when it moves normally.

The Bear Class (cont'd.)

```
def tryToEat(self):
    offsetList = [(-1,1) ,(0,1) ,(1,1),
                  (-1,0)        ,(1,0),
                  (-1,-1),(0,-1),(1,-1)]
    adjprey = []
    for offset in offsetList:
        newx = self.xpos + offset[0]
        newy = self.ypos + offset[1]
        if 0 <= newx < self.world.getMaxX() and
           0 <= newy < self.world.getMaxY():
            if (not self.world.emptyLocation(newx,newy)) and
                   isinstance(self.world.lookAtLocation(newx,newy),Fish):
                # adds the neighboring Fish to the list
                adjprey.append(self.world.lookAtLocation(newx,newy))
    # if there are any Fish adjacent to the bear, one is chosen at random
    # else, then starveTick is incremented
    if len(adjprey)>0:
        randomprey = adjprey[random.randrange(len(adjprey))]
        preyx = randomprey.getX()
        preyy = randomprey.getY()
        # delete the fish and tell the bear to move to the location formerly occupied by the fish.
        # Since the bear has now eaten, it is no longer starving and starveTick can be reset to zero
        self.world.delThing(randomprey)
        self.move(preyx,preyy)
        self.starveTick = 0
    else:
        self.starveTick = self.starveTick + 1
```

Main Simulation

- We can now turn our attention to implementing a function that will be responsible for setting up the entire simulation and starting it.
- The simulation itself dictates how it progresses due to the random numbers that are generated.
- This main function will need to do four things:
  1. Set initial constants pertaining to the number of bears and fish, the size of the world, and the length of the simulation.
  2. Create an instance of the World.
  3. Create the specified number of bears and fish and place them at random locations in the world.
  4. Let the world live for the specified number of time units.

Main Simulation (cont’d.)

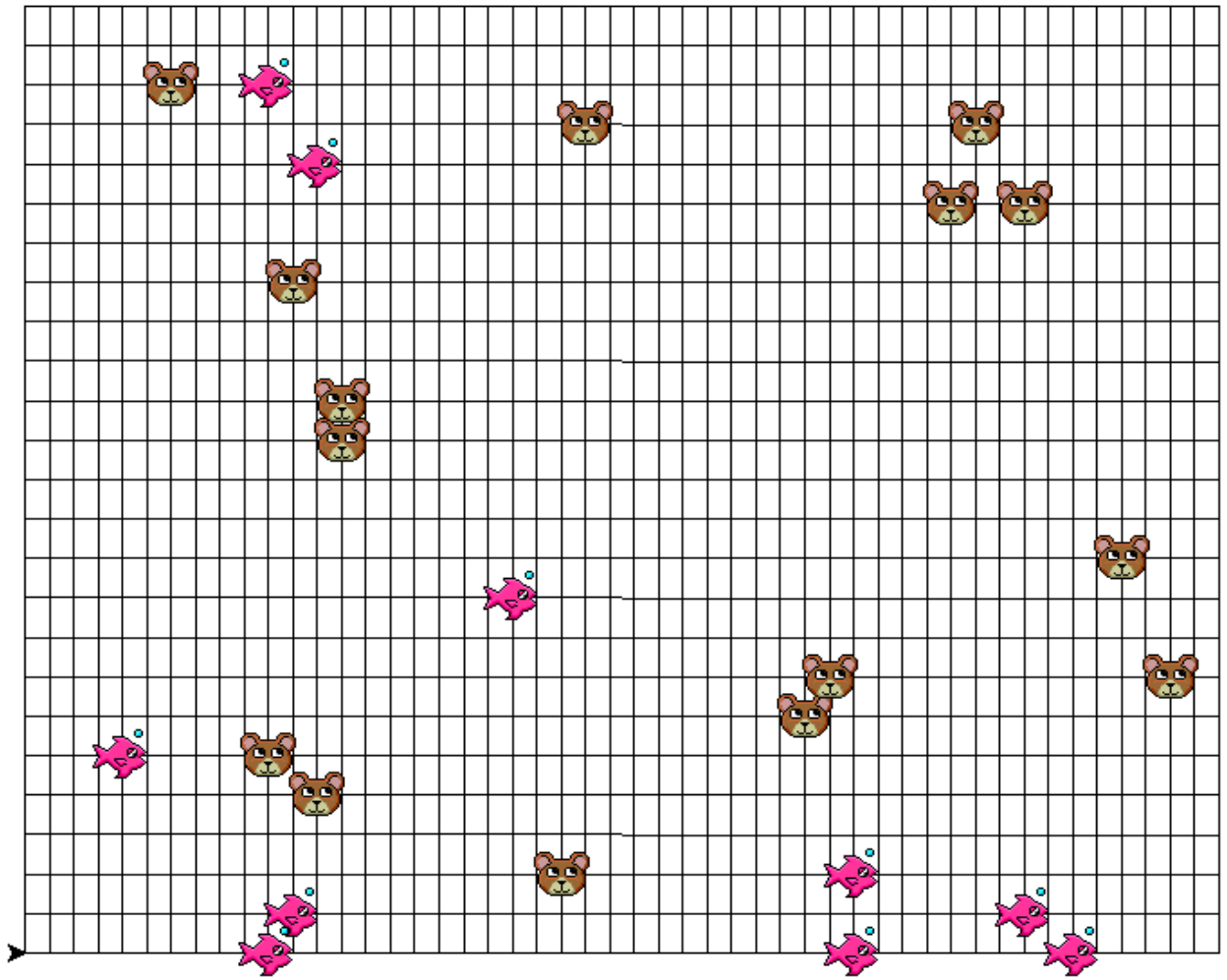
- Once a fish has been created, a random *(x, y)* location must be generated.
- Since we do not want more than one life-form living at a specific location, we must check to see if the location is empty before using it.
- If it is not empty, we continue to randomly generate random *(x, y)* locations.
- Bears are created and placed in the same way.
- For each time unit, my World's liveALittle method is called.
  - this means that a random life-form will be chosen to live for that time unit.

Main Simulation (cont'd.)

```
def mainSimulation():
    # initialize the world with ten bears and ten fish
    numberOfBears = 10
    numberOfFish = 10
    worldLifeTime = 2500
    worldWidth = 50
    worldHeight = 25
    # creates a World and draw it
    myworld = World(worldWidth,worldHeight)
    myworld.draw()
```

```
    # create and place the Fish objects in the world
    for i in range(numberOfFish):
        newfish = Fish()
        x = random.randrange(myworld.getMaxX())
        y = random.randrange(myworld.getMaxY())
        while not myworld.emptyLocation(x,y):
            x = random.randrange(myworld.getMaxX())
            y = random.randrange(myworld.getMaxY())
        myworld.addThing(newfish,x,y)
    # create and place the Bear objects in the world
    for i in range(numberOfBears):
        newfish = Bear()
        x = random.randrange(myworld.getMaxX())
        y = random.randrange(myworld.getMaxY())
        while not myworld.emptyLocation(x,y):
            x = random.randrange(myworld.getMaxX())
            y = random.randrange(myworld.getMaxY())
        myworld.addThing(newfish,x,y)
```

Main Simulation (cont'd.)



Growing Plants

- Fish could also be predators in that they could eat plants that exist in the water.
- If there are not enough plants available, the fish would starve and die.
- It is relatively easy to include another class to represent the plants by taking code that already exists and simply "repackaging" it as required for a plant.
- We can begin by realizing that plants will not be able to move. In addition, we will not have them eat anything.
- We will assume that plants can attempt to breed after five time units.

```
def liveALittle(self):
        self.breedTick = self.breedTick + 1
        if self.breedTick >= 5:
            self.tryToBreed()
```

Growing Plants (cont’d.)

- The only other change that we need to make is the inclusion of a tryToEat method as part of the Fish class. This method would be identical to the tryToEat method of the Bear class except for the one modification that fish eat plants.
- Finally, we need to modify the main function so that plants are also added to the simulation at random locations.

Growing Plants (cont'd.)

```
def tryToEat(self):
    offsetList = [(-1,1) ,(0,1) ,(1,1),
                  (-1,0)        ,(1,0),
                  (-1,-1),(0,-1),(1,-1)]
    adjprey = []
    for offset in offsetList:
        newx = self.xpos + offset[0]
        newy = self.ypos + offset[1]
        if 0 <= newx < self.world.getMaxX() and 0 <= newy < self.world.getMaxY():
            if (not self.world.emptyLocation(newx,newy)) and isinstance(self.world.lookAtLocation(newx,newy),Plant):
                adjprey.append(self.world.lookAtLocation(newx,newy))

    if len(adjprey)>0:
        randomprey = adjprey[random.randrange(len(adjprey))]
        preyx = randomprey.getX()
        preyy = randomprey.getY()

        self.world.delThing(randomprey)
        self.move(preyx,preyy)
        self.starveTick = 0
    else:
        self.starveTick = self.starveTick + 1
```

Growing Plants (cont'd.)

```
class Plant:
    def __init__(self):
        self.turtle = turtle.Turtle()
        self.turtle.up()
        self.turtle.hideturtle()
        self.turtle.shape("Plant.gif")

        self.xpos = 0
        self.ypos = 0
        self.world = None

        self.starveTick = 0
        self.breedTick = 0

    def setX(self,newx):
        self.xpos = newx

    def setY(self,newy):
        self.ypos = newy

    def getX(self):
        return self.xpos

    def getY(self):
```

```
def setY(self,newy):
    self.ypos = newy

def getX(self):
    return self.xpos

def getY(self):
    return self.ypos

def setWorld(self,aworld):
    self.world = aworld

def appear(self):
    self.turtle.goto(self.xpos, self.ypos)
    self.turtle.showturtle()

def hide(self):
    self.turtle.hideturtle()

def move(self,newx,newy):
    self.world.moveThing(self.xpos,self.ypos,newx,newy)
    self.xpos = newx
    self.ypos = newy
    self.turtle.goto(self.xpos, self.ypos)
```

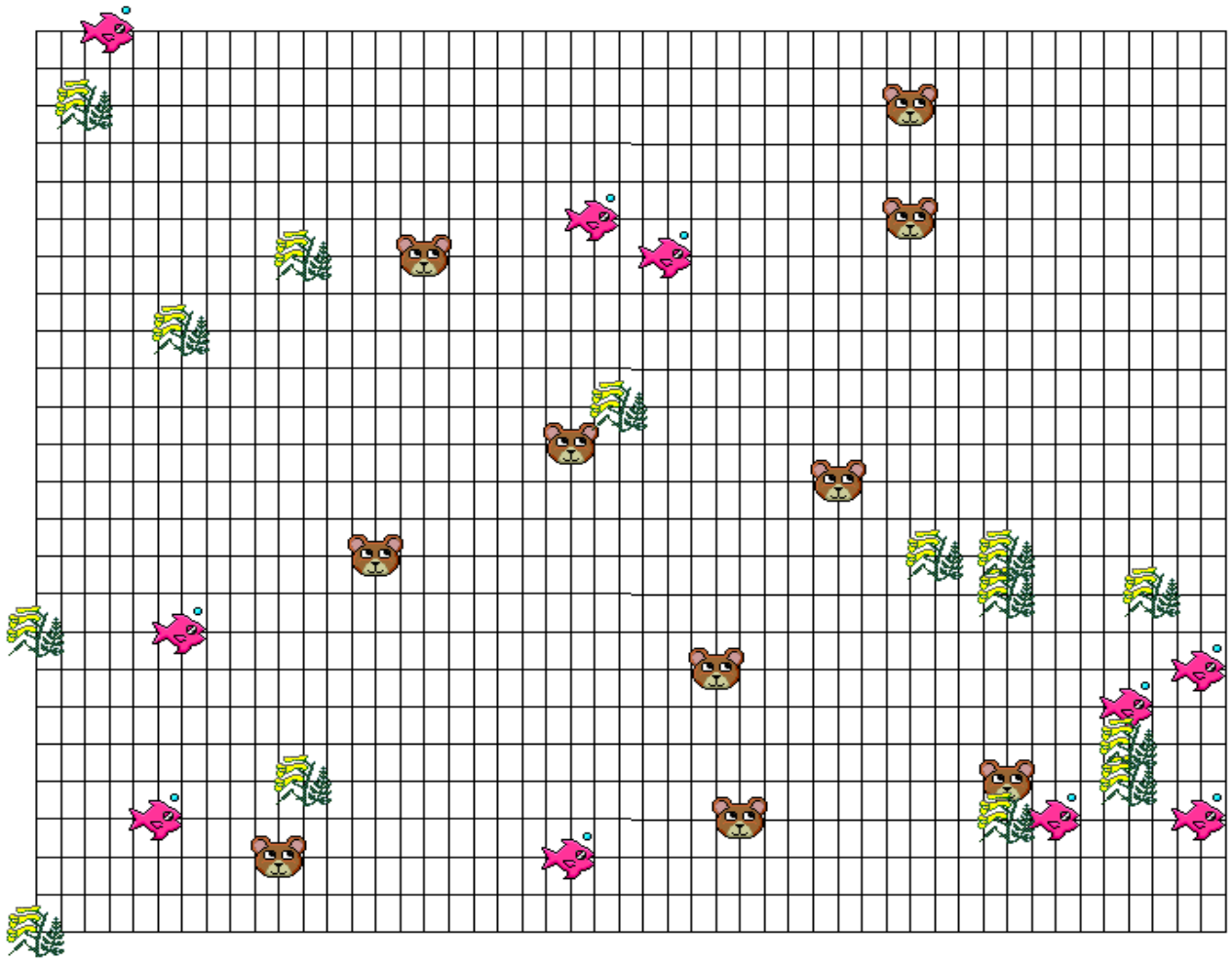
Growing Plants (cont'd.)

```
def liveALittle(self):
        self.breedTick = self.breedTick + 1
        if self.breedTick >= 5:
            self.tryToBreed()

def tryToBreed(self):
    offsetList = [(-1,1)  ,(0,1)  ,(1,1),
                  (-1,0)          ,(1,0),
                  (-1,-1),(0,-1),(1,-1)]
    randomOffsetIndex = random.randrange(len(offsetList))
    randomOffset = offsetList[randomOffsetIndex]
    nextx = self.xpos + randomOffset[0]
    nexty = self.ypos + randomOffset[1]
    while not (0 <= nextx < self.world.getMaxX() and
               0 <= nexty < self.world.getMaxY() ):
        randomOffsetIndex = random.randrange(len(offsetList))
        randomOffset = offsetList[randomOffsetIndex]
        nextx = self.xpos + randomOffset[0]
        nexty = self.ypos + randomOffset[1]

    if self.world.emptyLocation(nextx,nexty):
       childThing = Plant()
       self.world.addThing(childThing,nextx,nexty)
       self.breedTick = 0
```

Growing Plants (cont’d.)



Summary

- In this chapter we designed and implemented a large, multiclass graphical simulation.
- In order to design a class, we described those things that instances of a class would need to know about themselves (the instance variables) and those things that instances of a class would need to be able to do (the methods).
- After creating each class, we implemented a function that established many objects and let each one perform methods as defined by the initial rules of the simulation.
- By using random numbers, we were able to provide a degree of uncertainty so that no two runs of our simulation would look the same.