CHAPTER 12

Object-oriented Design

> What we desire from an abstraction is a mechanism which permits the expression of relevant details and the suppression of irrelevant details. In the case of programming, the use which may be made of an abstraction is relevant; the way in which the abstraction is implemented is irrelevant.
>
> Barbara Liskov
> *Programming with Abstract Data Types (1974)*

Our problem solving strategy to this point has focused on the decomposition of a problem into smaller subproblems, each viewed as a functional abstraction. We design algorithms, and then write functions, for these subproblems, and combine them to solve our overall problem.

An alternative design strategy, called object-oriented design, instead focuses on the data, or objects, in a problem. To solve a problem, we identify the objects involved, design ***abstract data types*** for them, and then implement them as ***classes***.

Recall that an ADT is defined by the information it can store, called ***attributes***, and a set of ***operations*** that can access that information. We have used a variety of ADTs, such as turtles, strings, lists, and dictionaries, each implemented as a class in Python. A class serves as a blueprint for a category of data. An ***object*** is a particular instance of a class. In Section 2.1, we described this difference by analogy to a species and the organisms that belong to that species. The species description is like a class; it describes a category of organisms but is not an organism itself. The individual organisms belonging to a species are like objects from the same class.

The attributes of a class are assigned to a set of variables called ***instance variables***. The operations of a class are special functions called ***methods***. Instance variables remain hidden to a programmer using the class, and are only accessed or modified

indirectly through methods. For example, the `Turtle` class contains several hidden instance variables that store each `Turtle` object's position, color, heading, and whether its tail is up or down. The `Turtle` class also defines several familiar methods, such as `forward/backward`, `left/right`, `speed`, and `up/down` that we can call to indirectly interact with these instance variables. When we create the two new `Turtle` objects named `george` and `diego` below, although they belong to the same class, they maintain independent identities because each instance (object) has its own copies of the `Turtle` instance variables.

```
george = turtle.Turtle()
diego = turtle.Turtle()
```

In this chapter, we will use object-oriented design to solve problems with new custom classes that behave the same way the built-in classes do. We will start by designing an object-oriented simulation of an epidemic virus. Then we will implement a more utilitarian class to illustrate all of the ways in which a class can be made to behave like the standard classes that you have been using all along. Then we will explore a more advanced simulation of flocking birds and the design of two new ADTs.

12.1 SIMULATING AN EPIDEMIC

Simulations are widely used to facilitate planning for large-scale epidemics, or a pandemic like COVID-19. These simulations can generally take one of two forms: a population model or an agent-based model. A population model treats a population as a group of identical individuals and is only concerned with the populations' sizes. The SIR model that you may have seen in Section 4.4 is the basis of most population models for viral epidemics like COVID-19. In contrast, an ***agent-based simulation*** contains a set of independent individuals (the agents) that interact with each other in some way over time. Schelling's model of racial segregation from Project 8.1 is a simple example of an agent-based simulation.

In this section, we will write an agent-based simulation of a viral epidemic using an object-oriented design. The main objects in the simulation are the agents (in our case, people) and a two-dimensional "world" in which the agents move. The world will also act as the glue that binds the agents together and drives the simulation by repeating the following simple algorithm for some length of time:

Algorithm EPIDEMIC SIMULATION STEP

```
1  repeat for each person in the world:
2      move the person one step forward
3      if the person is infected, then:
4          probabilistically infect every non-infected person within some distance
```

Object design

The first step in writing this simulation is to design the two objects as abstract data types, analogous to how we start a functional design by writing algorithms in pseudocode. When we design an ADT or write an algorithm, we are free to focus on the problem at hand, unencumbered by the requirements of the programming language. After this design phase, we will implement each of the ADTs as a class.

Reflection 12.1 *Based on the simulation algorithm above, what attributes and operations does a person object need?*

You can probably think of many possible attributes for a person in this simulation, but we will keep it simple at first. Every Person object will at least need a world to live in, a position and heading as they move in the world, and a variable that tracks whether they are infected with the virus.

Instance Variable	Description
world	the world inhabited by the person
position, heading	the person's current position and heading in the world
infected	whether the person is currently infected (Boolean)

Based on the simulation algorithm, every person will need the ability to move around (randomly) in the world and become infected if they come too close to an infected person. We will also need to be able to access attributes of people, such as their position, whether they are infected, and whether they are too close to another infected person. The following six operations will handle these basic needs.

Method	Arguments	Description
create	world, infected	create a new person with random position and heading in world and infect if infected is true
get position	—	return the person's position as a tuple
is infected	—	return whether the person is infected
within	person, distance	return true if a given person is within distance of my position, false otherwise
infect	infection probability	set infected to true with the given probability
step	—	take one step in the simulation

You may notice that these methods fall into three categories:

1. A *constructor* creates a new instance of an ADT.
2. An *accessor* reads the attributes of an instance and returns information derived from them, but does not modify the attributes' values.
3. A *mutator* modifies the values of the attributes of an instance.

Reflection 12.2 *To which category does each of the six* Person *methods belong?*

The first operation we defined is the constructor because it creates a new Person instance. The next three operations are accessors because they give information derived from the attributes of an instance without modifying it. Finally, infect and step are mutators because they may change the attributes of an instance.

Before we implement the Person class, let's also lay out the structure of the World ADT. The world will need dimentions, a list of the people in the world, the probability that a person becomes infected if they come into contact with an infected person, and the number of people infected. These are maintained in the following five attributes.

Instance Variable	Description
width, height	the width and height of the world
infection probability	the probability that a person becomes infected if they come into contact with an infected person
people	a list of people in the world
number infected	the number of infected people

In addition, based on the simulation algorithm, we know that the World will need to able to infect people who come too close to an already-infected person, and run one step of the main simulation loop, which we call step all. We will also need some accessor methods to get attributes of the world when needed.

Method	Arguments	Description
create	width, height, infection probability, population size	create a new world with the given dimensions and infection probability, and populate it with one infected person and population size – 1 uninfected people
get width, get height	—	return the width and height
get number infected	—	return number infected
infect neighbors	person	infect neighbors of an infected person with probability = infection probability
step all	—	move all inhabitants one step and spread the infection

Person class

Let's begin our implementation by designing a class that implements a simplified version of the Person ADT, one that is not tied to any World. This will allow us to experiment and get a better feel for how classes work, before we dive into the complete simulation.

The constructor

The definition of a new class begins with the keyword `class` followed by the name of the class and, of course, a colon. The class' methods are indented below.

The constructor of a class is named `__init__` (with two underscore characters at both the beginning and the end). The beginning of the `SimplePerson` class, with its constructor, is shown below.

```
class SimplePerson:
    """A simple person class."""

    _STEP = 5   # class variable (constant)

    def __init__(self, infected):
        """Create a new, possibly infected, person with random heading."""

        self._infected = infected
        self._turtle = turtle.Turtle()
        self._turtle.setheading(random.randrange(360))

        if self._infected:
            self._turtle.color('red')
        else:
            self._turtle.color('blue')
```

The constructor is *implicitly* called when we create a new object by calling the function bearing the name of the class. For example, when we created the two `Turtle` objects above, we implicitly invoked the `Turtle` constructor twice. To invoke the constructor of the `SimplePerson` class to create a new, uninfected `SimplePerson` object, we could call

```
someone = SimplePerson(False)
```

The first parameter of the `__init__` method, named `self`, is an *implicit* reference to the object on which the method is being called. In the assignment statement above, `self` is assigned to the new object being created. This same object is returned by the constructor and assigned to `someone`. We never *explicitly* pass anything in for `self`. The additional constructor parameter `infected` is a Boolean value used to initialize the object's infection status.

The `SimplePerson` class has two instance variables named `self._infected` and `self._turtle`. The former is a Boolean value indicating whether the person is currently infected with the virus and the latter is a visual representation of the person in the simulation. We will also use `self._turtle` to implicitly store each person's position and heading (since it will do that anyway).

Every instance variable name is preceded by `self` to signify that it belongs to the particular instance (object) of the class assigned to `self`. For example, since `self` is assigned to the new object created by the constructor, the assignment statement

```
someone = SimplePerson(False)
```

is creating a new `SimplePerson` object named `someone` and assigning values to `someone._infected` and `someone._turtle`.

The underscore (`_`) character before each instance variable name is a Python convention that indicates that the instance variables should be *private*, i.e., never accessed from outside the class.[1] We want instance variables to be private so they can only be changed by methods of the class, and not in unintended ways outside the class. For example, we do not want a simulation using our class to incorrectly move a person's turtle or change their infection status in ways that might mess up the simulation. This idea, which is a key characteristic of object-oriented programming, is called ***encapsulation***. Encapsulation also refers more generally to the practice of bundling instance variables and methods together in a class.

The scope of an instance variable is the *entire object*. This means that we can access and change the value of any instance variable in any method of the class. In contrast, variable names defined inside a method that are not preceded by `self`, such as the `infected` parameter in the constructor, are just normal local variables with scope limited to the method.

Just before the constructor above is a ***class variable*** named `_STEP`, which will act as a constant move distance for all people turtles. A class variable is shared by all objects in a class. Since `_STEP` is a constant, it doesn't make sense to have a separate copy for every object; instead, all objects will share this one copy.

Accessor methods

Now let's add the three accessor methods to the `SimplePerson` class.

```
    def getPosition(self):
        """Return the person's position as a tuple.

        Parameter:
            self: the Person object

        Return value: position of self as a tuple
        """

        return self._turtle.position()

    def isInfected(self):
        """Return whether the person is infected.

        Parameter:
            self: the Person object

        Return value: Boolean indicating whether self is infected
        """

        return self._infected
```

[1] Python does not actually enforce this, but some other languages do.

```
    def within(self, otherPerson, distance):
        """Return True if otherPerson is within distance
           of my position, False otherwise.

        Parameters:
            self:        the Person object
            otherPerson: another Person object
            distance:    a number

        Return value: Boolean indicating whether otherPerson's position
                      is within distance of self's position
        """

        myPosition = self.getPosition()
        otherPosition = otherPerson.getPosition()
        diffX = myPosition[0] - otherPosition[0]
        diffY = myPosition[1] - otherPosition[1]
        return math.sqrt(diffX ** 2 + diffY ** 2) <= distance
```

The `getPosition` method uses the `position` method of `self._turtle` to return a tuple containing the current position. The only parameter to this method is `self`. If we called `someone.getPosition()`, the object `someone` is implicitly passed in for the parameter `self`, even though it is not passed in the parentheses following the name of the method. Similarly, the `isInfected` method simply returns the value of `self._infected`. The `within` method computes the distance between `self` and another `SimplePerson` object and returns a Boolean value indicating whether they are within the given `distance` of each other. Notice that the method calls the `getPosition` method of both `self` and `otherPerson` to get tuples of their positions. When we call a method of the class from within another method, we still need to preface the name of the method with `self` or another object, just as we do with instance variables.

Mutator methods

To round out the class, we will add the following two mutator methods.

```
    def infect(self, infectionProbability):
        """Infect self with the given probability.

       Parameters:
            self:                the Person object
            infectionProbability: probability of infection

        Return value: Boolean indicating whether infection happened
        """

        if not self._infected and random.random() < infectionProbability:
            self._infected = True
            self._turtle.color('red')
            return True

        return False
```

```
    def step(self):
        """Advance self one step in the simulation.

        Parameter:
            self: the Person object

        Return value: None
        """

        if random.random() < 0.1:
            self._turtle.left(random.randrange(-90, 90))
        self._turtle.forward(self._STEP)
```

The `infect` method infects the person, if they are not already infected, with the given probability. If the person is infected, the method returns `True` to signify "success." If the object is not infected, it returns `False`. In the `step` method, we simulate a person's movement by normally (90% of the time) moving forward along their current heading, and occasionally (10% of the time) turning to the left or right by some random angle.

When we write classes, we will store each one in its own file. By convention, the names of our classes will be capitalized, but the filenames will be in lowercase.

Reflection 12.3 *Create a new file named* `simpleperson.py` *containing the* `SimplePerson` *class. (You will also need to* `import` *some modules at the top.)*

After saving the `SimplePerson` class in `simpleperson.py`, we can create a new, uninfected `SimplePerson` object with

```
>>> import simpleperson
>>> someone = simpleperson.SimplePerson(False)
```

or

```
>>> from simpleperson import *
>>> someone = SimplePerson(False)
```

Reflection 12.4 *Create a new* `SimplePerson` *object in a Python shell with one of the options above. Or you can write a short program in the same directory as* `simpleperson.py` *if you have trouble importing from the shell. What happens when create the object?*

When you create a new `SimplePerson` object, a turtle graphics window should open and display a blue turtle facing in a random direction in the center of the screen. This is `someone._turtle`, the `Turtle` object inside the `someone` object. Now call the `getPosition` method, followed by a few calls to `step`.

```
>>> someone.getPosition()
(0.00,0.00)
>>> someone.step()
>>> someone.step()
>>> someone.getPosition()
(9.21,-3.91)  # your result will differ
```

Each time you call `step`, you are invoking the `step` method on the `SimplePerson` object named `someone`, which moves `someone`'s turtle a little. You can see how this has changed `someone`'s `_turtle` instance variable when you call `getPosition` again. Since `step` moves the turtle so little, try calling it in a loop:

```
>>> for count in range(50):
        someone.step()
```

Initially, `someone` is not infected with the virus (because we passed `False` into the constructor), which you can verify by calling the `isInfected` method.

```
>>> someone.isInfected()
False
```

Now try infecting `someone` with probability 0.5. You may have to try a few times until it is successful. Then verify that it worked by calling `isInfected` again.

```
>>> someone.infect(0.5)
False
>>> someone.infect(0.5)
True
>>> someone.isInfected()
True
```

After `someone` becomes infected, you should notice that the turtle turns red. Next create another, uninfected `SimplePerson` object and move them a bit.

```
>>> someoneElse = SimplePerson(False)
>>> someoneElse.isInfected()
False
>>> for count in range(50):
        someoneElse.step()
```

If we want to know if this new person is within some distance of the infected `someone`, we can call the `within` method.

```
>>> someoneElse.within(someone, 10)
False
>>> someoneElse.within(someone, 500)
True
```

When you call the `within` method in this way, `someoneElse` is passed in for `self` and `someone` is passed in for `otherPerson`. Chances are, they are not very close to each other but if you keep trying larger distances, the method should eventually return `True`.

Reflection 12.5 *Does calling* `someone.within(someoneElse, 500)` *do the same thing? In this case, which object is assigned to* `self` *and which is assigned to* `otherPerson`*?*

Augmenting the Person class

Now that you are a more comfortable with the mechanics of classes, let's flesh out the full `Person` class that we will use in our simulation. The `Person` class will be identical to the `SimplePerson` class, except for edits and additions to two methods. The first changes are to the constructor, highlighted below.

```
class Person:
    """A person in an epidemic simulation."""

    _STEP = 5   # class variable (constant)

    def __init__(self, myWorld, infected):
        """Create a person with random position/heading in myWorld."""

        self._world = myWorld
        self._infected = infected

        self._turtle = turtle.RawTurtle(self._world._screen)
        self._turtle.speed(0)
        self._turtle.up()
        self._turtle.resizemode('user')
        self._turtle.shape('circle')
        self._turtle.shapesize(0.5)

        self._turtle.setheading(random.randrange(360))
        x = random.randrange(self._world.getWidth())
        y = random.randrange(self._world.getHeight())
        self._turtle.goto(x, y)

        if self._infected:
            self._turtle.color('red')
        else:
            self._turtle.color('blue')
```

First, we added a `myWorld` parameter that will serve as the `World` object to which the `Person` belongs. (We will implement `World` next.) We have also assigned `_turtle` to a `RawTurtle` object instead of a normal `Turtle` object. `RawTurtle` is just like `Turtle`, but it will allow us to do some fancier graphical interface things later. The `TurtleScreen` object named `self._world._screen` that we pass into the `RawTurtle` constructor is an instance variable of the `World` class. Its purpose is to make sure that every `Person` turtle draws in the same window. We also added some turtle formatting that will make each person a small circle. Finally, we give each person a random starting position, using the soon-to-be-implemented `getWidth` and `getHeight` methods of the `World` class to set the bounds of the position.

The second edit is to the `step` method as highlighted below.

```
    def step(self):
        """ (docstring omitted) """

        if random.random() < 0.1:
            self._turtle.left(random.randrange(-90, 90))
        self._turtle.forward(self._STEP)

        # wrap around to the other side of the world if necessary
        newX = self._turtle.xcor() % self._world.getWidth()
        newY = self._turtle.ycor() % self._world.getHeight()
        if self._turtle.position() != (newX, newY):
            self._turtle.goto(newX, newY)
```

This addition guards against a person stepping off the edge of the world. When this happens, rather than have them "bounce" back, we wrap them around to the other side by using modular arithmetic. In this way, the world is treated like a torus.

Reflection 12.6 *Create a new file named* `person.py` *containing the* `Person` *class. Start from the* `SimplePerson` *class and make the highlighted changes.*

World class

The constructor of the `World` class will take five parameters, in addition to `self`.

```
from person import *

class World:
    """A two-dimensional world class."""

    _INFECT_DISTANCE = 6   # class variable (constant)

    def __init__(self, width, height, infectProb, popSize, screen):
        """Create a new world with the given dimensions and infection
           probability, and populate it with one infected person and
           popSize - 1 uninfected people."""

        self._width = width
        self._height = height
        self._infectionProbability = infectProb
        self._screen = screen

        self._numberInfected = 1
        self._people = [Person(self, True)]  # one infected person

        for index in range(popSize - 1):     # uninfected people
            person = Person(self, False)
            self._people.append(person)
```

At the top of the file, we need to `import` the `Person` class so that we can create new people in the constructor. Just before the constructor is a class variable named `_INFECT_DISTANCE` which is how close someone needs to be to an infected person to become infected themselves.

Reflection 12.7 *How many instance variables does the* `World` *class have?*

The class has six instance variables. The first four, `self._width`, `self._height`, `self._infectionProbability` and `self._screen`, are initialized by parameters. The `_screen` instance variable is the name of the `TurtleScreen` object on which all of the `Person` turtles will be drawn. It will be created by the main program. The remainder of the constructor populates the world by creating a list named `self._people` containing one infected person and `popSize - 1` uninfected people. The `Person` constructor is called to create each person.

In addition to the constructor, the `World` class will have the following three accessor methods.

```
    def getWidth(self):
        """ (docstring omitted) """

        return self._width

    def getHeight(self):
        """ (docstring omitted) """

        return self._height

    def getNumberInfected(self):
        """ (docstring omitted) """

        return self._numberInfected
```

As in the `Person` class, these methods simply return the values of instance variables so that the instance variables are never accessed directly from outside the class. The `getWidth` and `getHeight` methods are used in the `step` method of the `Person` class, and the `getNumberInfected` method will be used by our main program to plot the number of infected individuals over the course of the simulation.

Finally, the `World` class is rounded out by two mutator methods.

```
    def infectNeighbors(self, infectedPerson):
        """ (docstring omitted) """

        for otherPerson in self._people:
            if not otherPerson.isInfected() and \
              otherPerson.within(infectedPerson, self._INFECT_DISTANCE):
                if otherPerson.infect(self._infectionProbability):
                    self._numberInfected = self._numberInfected + 1

    def stepAll(self):
        """ (docstring omitted) """

        for person in self._people:
            person.step()
            if person.isInfected():
                self.infectNeighbors(person)
```

The `infectNeighbors` method takes an infected person as a parameter and then iterates over everyone in the world, infecting anyone within `self._INFECT_DISTANCE` with probability `self._infectionProbability`. If a person is successfully infected, the value of `self._numberInfected` is incremented. The `stepAll` method implements each step of the simulation, as we laid out at the beginning of the section.

Reflection 12.8 *Create another new file named* `world.py` *containing the* `World` *class.*

The simulation

With our classes created, the following program will drive the simulation.

```
import turtle
from world import *

WIDTH = 600           # width of the world
HEIGHT = 600          # height of the world
NUM_PEOPLE = 200      # number of people to simulate
INFECTION_PROB = 0.5  # probability that someone gets infected

def main():
    worldScreen = turtle.Screen()     # a screen for the turtles
    worldScreen.setup(WIDTH, HEIGHT)  # set window size
    worldScreen.setworldcoordinates(0, 0, WIDTH - 1, HEIGHT - 1)
    worldScreen.tracer(0)             # turn off screen updates

    world = World(WIDTH, HEIGHT, INFECTION_PROB, NUM_PEOPLE, worldScreen)

    while world.getNumberInfected() < NUM_PEOPLE:  # until all infected
        world.stepAll()       # advance all people one step
        worldScreen.update()  # manually update screen after each step

    worldScreen.exitonclick()

main()
```

The `main` function creates a `Screen` object named `worldScreen` on which the turtles representing people can live. This is passed in as the last parameter of the `World` constructor. Then the program iterates until the number of infected people is equal to the total number of people. In each iteration, the simulation is advanced one step by calling `world.stepAll()`. A screenshot of the finished simulation is shown on the lefthand side of Figure 12.1.

Reflection 12.9 *Augment the* `main` *function so that it plots the number of infected people over the course of the simulation. Display your plot just before* `worldScreen.exitonclick()`. *It should look similar to the plot on the righthand side of Figure 12.1.*

On the book website you can find an augmented version of this program that incorporates sliders for the number of people and infection probability, and plots the number infected as the simulation is running. A screenshot is shown in Figure 12.2. This program also demonstrates how to use the graphics framework underlying turtle graphics, called `Tkinter`, to add graphical user interface elements to programs. Later in this chapter, we will design a more sophisticated agent-based simulation of flying birds in which each bird interacts with other birds in the flock, resulting in emergent flocking behavior.

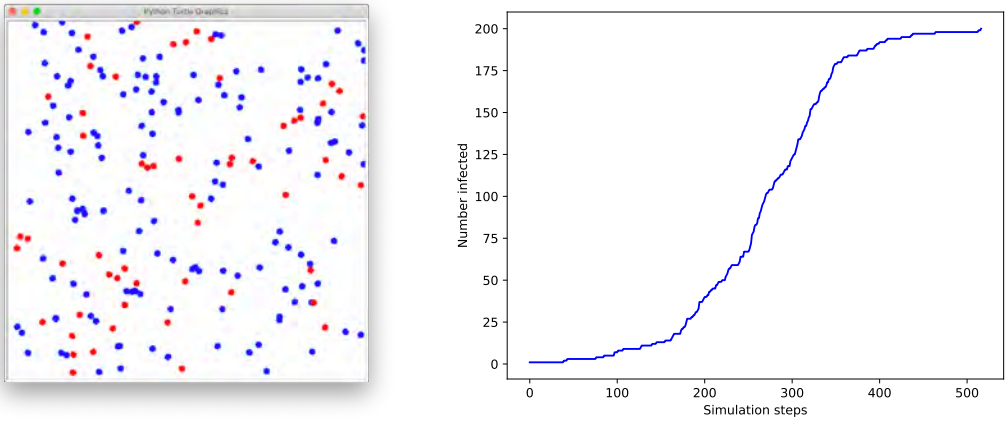


Figure 12.1 On the left is a screenshot midway through the epidemic simulation with 200 people and infection probability 0.5. On the right is a plot of the number infected.

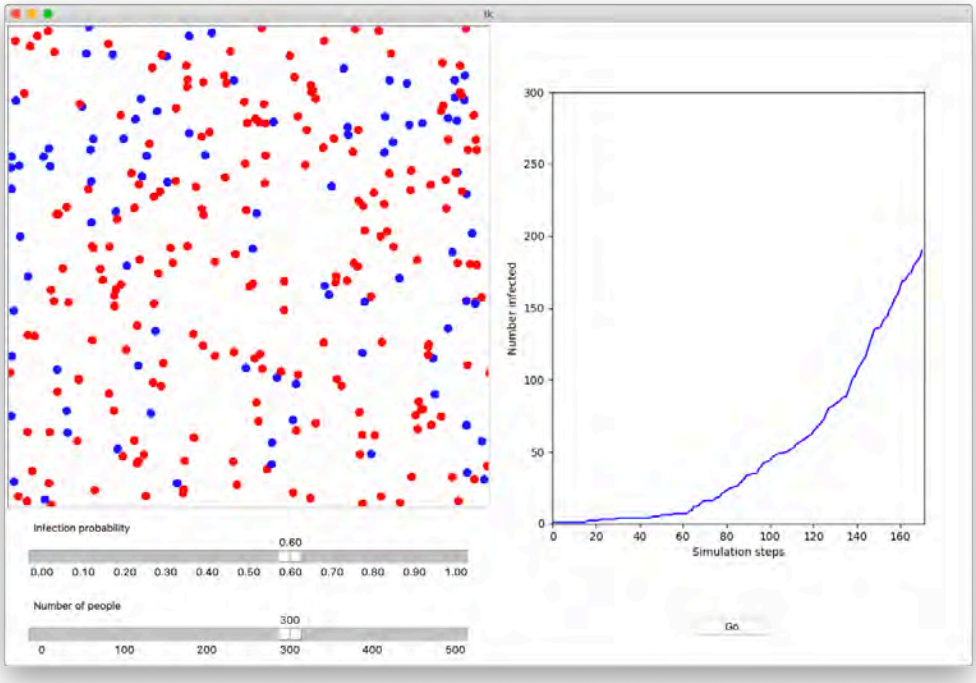


Figure 12.2 A screenshot midway through the augmented epidemic simulation.

Exercises

12.1.1* Name two accessor methods and two mutator methods in the `Turtle` class.

12.1.2. Name two accessor methods and two mutator methods in the `list` class.

12.1.3. Add a new method

```
allInfected(self)
```

to the `World` class that returns `True` if everyone in the world is infected, and `False` otherwise. Show how to use this new method in the `while` loop of the main simulation.

12.1.4* Add a new method

```
add(self, person)
```

to the `World` class that adds a new `Person` object named `person` to the world. If the person is infected, increment the value of `self._numberInfected`.

12.1.5. Add a new method

```
distance(self, otherPerson)
```

to the `Person` class that returns the distance between the `Person` objects `self` and `otherPerson`. Show how to use your new method to simplify the `within` method.

12.1.6. In this exercise, you will modify the epidemic simulation so that some people stay at home during the epidemic.

(a) Add a new instance variable to the `Person` class named `self._home`, initialized to `False`, which will indicate whether the person is sheltered at home.

(b) Add a method named `stayHome` to the `Person` class that sets `self._home` to `True`.

(c) Modify the `infect` method of the `Person` class so that a person at home cannot become infected.

(d) Modify the `step` method of the `Person` class so that the person does not move if they are at home.

(e) Add a parameter to the constructor of the `World` class that defines the probability that a person will stay home. In the loop that populates the world, call `person.stayHome()` with that probability.

(f) Run the simulation with these modifications. You will need to modify the loop in the `main` function so that it runs for a particular number of iterations (say, 1000) since now the entire population is unlikely to get infected all at once. What do you notice from the plot? (This assumes you have completed Reflection 12.9.)

12.1.7. In this exercise, you will modify the epidemic simulation so that infected people can recover and become immune after a specified number of simulation steps.

(a) Add two new instance variables to the `Person` class named `self._infectedSteps` and `self._immune`. The former counts the number of simulation steps that have elapsed since the person has been

infected. The second is a Boolean value indicating whether the person is immune to the virus.

(b) Add a method named `isImmune` to the `Person` class that returns the value of `self._immune`.

(c) Modify the `infect` method of the `Person` class so that a person cannot become infected if they are immune.

(d) Add a method named `isRecovered` to the `Person` class that checks if the person is infected and, if so, checks whether `self._infectedSteps` is equal to a constant class variable named `_INFECTION_PERIOD`. If this is the case, the person becomes infected so set the person's `self._infected` and `self._immune` instance variables appropriately and their turtle's color to yellow. If the person is infected but has not yet been so for `self._INFECTION_PERIOD` steps, the method should increment the value of `self._infectedSteps`. The method should return `True` if the person becomes newly immune or `False` otherwise.

(e) Modify the `stepAll` method of the `World` class so that it calls `isRecovered` for every infected person in each iteration of the loop. If `isRecovered` returns `True`, then decrement the value of `self._numberInfected`.

(f) Run the simulation with your modifications and the class variable `_INFECTION_PERIOD` set to 100. You will need to modify the loop as specified in part (f) of the previous exercise. What do you notice from the plot? (This assumes you have completed Reflection 12.9.)

12.1.8. Design a research question you would like to investigate using the original epidemic simulation or the modified simulations from the previous two exercises. Run the simulation with various parameters to answer your question.

12.1.9* (a) Write a `BankAccount` class that has a single instance variable (the available balance), a constructor that takes the initial balance as a parameter, and methods `getBalance` (which should return the amount left in the account), `deposit` (which should deposit a given amount into the account), and `withdraw` (which should remove a given amount from the account).

(b) Using your `BankAccount` class from part (a), write a program that prompts for an initial balance, creates a `BankAccount` object with this balance, and then repeatedly prompts for deposits or withdrawals. After each transaction, it should update the `BankAccount` object and print the current balance. For example:

```
Initial balance? 100
(D)eposit, (W)ithdraw, or (Q)uit? d
Amount = 50
Your balance is now $150.00
(D)eposit, (W)ithdraw, or (Q)uit? w
Amount = 25
Your balance is now $125.00
(D)eposit, (W)ithdraw, or (Q)uit? q
```

12.1.10. (a) Write a class that represents a U.S. president. The class should include instance variables for the president's name, party, home state, religion, and age when he or she took office. The constructor should initialize the president's name to a parameter value, but initialize all other instance variables to default values (empty strings or zero). Write accessor and mutator methods for all five instance variables.

(b) On the book website is a tab-separated file containing a list of all U.S. presidents with the five instance variables from part (a). Write a function that reads this information and returns a list of president objects representing all of the presidents in the file. Also, write a function that, given a list of president objects and an age, prints a table with all presidents who where at least that old when they took office, along with their ages when they took office.

12.1.11. Write a `Movie` class that has as instance variables the movie title, the movie year, and a list of actors (all of which are initialized in the constructor). Write accessor and modifier functions for all the instance variables and an `addActor` method that adds an actor to the list of actors in the movie. Finally, write a method that takes as a parameter *another* movie object and checks whether the two movies have any common actors.

There is a program on the book website with which to test your class. The program reads actors from a movie file (like those used in Project 11.3), and then prompts for movie titles. For each movie, you can print the actors, add an actor, and check whether the movie has actors in common with another movie.

12.1.12. (a) Write a class representing a U.S. senator. The `Senator` class should contain instance variables for the senator's name, political party, home state, and a list of committees on which they serve. The constructor should initialize all of the instance variables to parameter values, except for the list of committees, which should be initialized to an empty list. Add accessor methods for all four instance variables, plus a mutator method that adds a committee to a senator's list of committees.

(b) On the book website is a function that reads a list of senators from a file and returns a list of senator objects, using the `Senator` class that you wrote in the previous exercise. Write a program that uses this function to create a list of `Senator` objects, and then iterates over the list of `Senator` objects, printing each senator's name, party, and committees. Then your program should prompt repeatedly for the name of a committee, and print the names and parties of all senators who are on that committee.

12.1.13. Write a class named `Student` that has the following instance variables: student name, exam grades, quiz grades, lab grades, and paper grades. The constructor should only take the student name as a parameter, but initialize all the other instance variables (to empty lists). Write an accessor method for the name and methods to add grades to the lists of exam, quiz, paper, and lab grades. Next, write methods for returning the exam, quiz, paper, and lab averages. Finally, write a method to compute the final grade for the course, assuming the average exam grade is worth 50%, the average quiz grade is worth 10%, and the average lab and paper grades are worth 20% each.

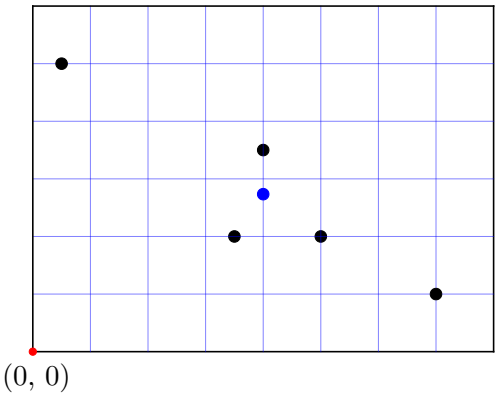
12.1.14* Write a class that represents a set of numerical data from which simple descriptive statistics can be computed. The class should contain five methods, in addition to the constructor: add a new value to the data set, return the minimum and maximum values in the data set, return the average of the values in the data set, and return the size of the data set. Think carefully about the instance variables needed for this class. It is not actually necessary for the class to include a list of all of the values that have been added to it.

12.1.15. This exercise assumes you read Section 6.8. Write a `Sequence` class to represent a DNA, RNA, or amino acid sequence. The class should store the type of sequence, a sequence identifier (or accession number), and the sequence itself. Identify and implement at least three useful methods, in addition to the constructor.

12.2 OPERATORS AND POLYMORPHISM

Suppose you are on the planning commission for your local town, and are evaluating possible locations for a new high school. One consideration is how central the new school will be with respect to homes within the district. If you know the location of each home, then you can compute the most central location, called the *centroid*, with respect to the homes. The centroid is the point whose x and y coordinates are the average of the x and y coordinates of the homes. (You may recall centroids from Section 7.7.)

For example, the five black points below might represent five houses, each with (x, y) coordinates representing the east-west and north-south distances (in km), respectively, from the point (0,0). The centroid of these points is shown in blue.



If the points are represented by a list of tuples like

```
homes = [(0.5, 5), (3.5, 2), (4, 3.5), (5, 2), (7, 1)]
```

then the following function can be used to return the centroid. (We will use abbreviated docstrings to save space.)

```
def centroid(points):
    """Compute the centroid of a list of points stored as tuples."""

    n = len(points)
    if n == 0:
        return None

    sumX = 0
    sumY = 0
    for point in points:
        sumX = sumX + point[0]
        sumY = sumY + point[1]
    return (sumX / n, sumY / n)
```

Calling `centroid(homes)` returns the tuple `(4, 2.7)`.

We can simplify working with points by designing a new, general-purpose *ordered pair* class. In addition to a geographic location, we could use this class to represent the (x,y) position of a particle, a (row, column) position in a grid, or a vector in two dimensions. The goal of this design will be to create a new utility class that behaves as if it was one of the standard Python classes. To that end, we want to be able to do arithmetic with pairs, print them, compare them, and even using indexing to access the individual elements.

Designing a Pair ADT

Let's begin by designing an ADT for this class. The obvious attributes are the two numbers, which we will simply name a and b.

Instance Variable	Description
a	the pair's first value
b	the pair's second value

Reflection 12.10 *What methods do we need for our pair ADT if we want to use it to compute centroids?*

The `centroid` function added points and divided by a scalar value, so we at least need those two operations. We will also need a constructor, and we should include methods to access and change the numbers in the pair. These operations are summarized in the table below.

Method	Arguments	Description
create	a and b	create a new pair instance (a, b)
getFirst	—	return the first value of the pair
getSecond	—	return the second value of the pair
get	—	return a tuple (a, b) representing the pair
add	pair 2	return a new pair that is the sum of this pair and pair 2
set	a and b	set new a and b values of the pair
scale	a number	multiply the values of a and b in the pair by number

Pair class

Let's now implement the Pair abstract data type as a class. We will start with the constructor and four other straightforward methods.

```
class Pair:
    """An ordered pair class."""

    def __init__(self, a = 0, b = 0):
        """Create a new Pair object initialized to (a, b)."""

        self._a = a   # the pair's first value
        self._b = b   # the pair's second value

    def getFirst(self):
        """ (docstring omitted) """

        return self._a

    def getSecond(self):
        """ (docstring omitted) """

        return self._b

    def get(self):
        """ (docstring omitted) """

        return (self._a, self._b)

    def set(self, a, b):
        """ (docstring omitted) """

        self._a = a
        self._b = b
```

Reflection 12.11 *Create a new file* `pair.py` *containing this class.*

The `= 0` following each of the `a` and `b` parameters in the constructor is specifying a *default argument*. This allows us to call the constructor with either no arguments,

in which case 0 will be assigned to `a` and `b`, or with explicit arguments for `a` and `b` that will override the default arguments. If we supply only one argument, then `a` will be assigned to it, and 0 will be assigned to `b`. For example:

```
pair1 = Pair()        # pair1 will represent (0, 0)
pair2 = Pair(3)       # pair2 will represent (3, 0)
pair3 = Pair(3, 14)   # pair3 will represent (3, 14)
```

Reflection 12.12 *How would you create a new* `Pair` *object with value* $(0, 18)$*?*

Arithmetic methods

We define the sum of two pairs (a, b) and (c, d) as the pair $(a+c, b+d)$. For example, $(3, 8) + (4, 5) = (7, 13)$. If we represented pairs as tuples, then an addition function would look like this:

```
def add(pair1, pair2):
    """Return a tuple representing the sum of tuples pair1 and pair2."""

    return (pair1[0] + pair2[0], pair1[1] + pair2[1])
```

To use this function to find `(3, 8) + (4, 5)`, we could do the following:

```
duo1 = (3, 8)
duo2 = (4, 5)
sumPair = add(duo1, duo2)   # sumPair is assigned (7, 13)
print(sumPair)              # prints "(7, 13)"
```

In an analogous `add` *method* for the `Pair` class, one of the points will be assigned to `self` and the other will be assigned to a parameter, as shown below.

```
    def add(self, pair2):
        """Return a new Pair that is the sum of Pairs self and pair2."""

        sumA = self._a + pair2._a
        sumB = self._b + pair2._b
        return Pair(sumA, sumB)
```

Notice that the method creates and returns a *new* `Pair` object. To find the sum of two `Pair` objects named `duo1` and `duo2`, we could call this method as follows:

```
duo1 = Pair(3, 8)
duo2 = Pair(4, 5)
sumPair = duo1.add(duo2)   # sumPair is assigned Pair(7, 13)
print(sumPair.get())       # prints "(7, 13)"
```

When the `add` method is called, `duo1` is assigned to `self` and `duo2` is assigned to `pair2`.

Reflection 12.13 *Add the* `add` *method to your* `Pair` *class. Then, define two new* `Pair` *objects in a* `main` *function and compute their sum. Because you may wish to import this module in the future, be sure to call* `main` *like this:*

```
if __name__ == '__main__':
    main()
```

Reflection 12.14 *Using the* `add` *method as a template, write a* `subtract` *method that subtracts another* `Pair` *object from* `self`*.*

In contrast to `add`, the `scale` method, as we defined it above, will modify the existing object rather than create a new one:

```
def scale(self, scalar):
    """Multiply the values in self by a scalar value."""

    self.set(self._a * scalar, self._b * scalar)
```

The `scale` method takes a numerical value (called a *scalar* in mathematics) as a parameter and uses the `set` method to multiply it by the values of `self._a` and `self._b`.

Reflection 12.15 *How could you write the* `scale` *method without calling* `self.set`*?*

Reflection 12.16 *Add the* `scale` *method to your class and use it on some* `Pair` *objects in your* `main` *function.*

Let's now revisit the `centroid` function, but modify it to handle a list of `Pair` objects instead of a list of tuples. The changes are highlighted below in red.

```
def centroid(points):
    """Compute the centroid of a list of Pair objects."""

    n = len(points)
    if n == 0:
        return None

    sumPair = Pair()                    # sumPair is the Pair (0, 0)
    for point in points:
        sumPair = sumPair.add(point)    # sumPair = sumPair + point
    sumPair.scale(1 / n)                # divide sumPair by n
    return sumPair
```

We have replaced the two `sumX` and `sumY` variables with a single `sumPair` variable, initialized to the pair $(0, 0)$. Inside the `for` loop, each value of `point`, which is now a `Pair` object, is added to `sumPair` using the `add` method. After the loop, we use the `scale` method to multiply the point by `1 / n` which, of course, is the same as dividing by `n`.

To use this function on the `homes` list from earlier, we would need to assign `homes` to be a list of `Pair` objects instead of tuples.

```
homes = [Pair(0.5, 5), Pair(3.5, 2), Pair(4, 3.5), Pair(5, 2), Pair(7, 1)]
central = centroid(homes)  # central is a Pair object
print(central.get())
```

Printing the value of the centroid is slightly more cumbersome because we have to convert it to a tuple first with the `get` method, but we will fix that shortly.

Reflection 12.17 *Add the code above to your* `main` *function. What is the value of the centroid?*

Special methods

The `centroid` method would be even more elegant if we could simply add `Pair` objects with the `+` operator. We have already seen how the `+` operator can be used with a variety of different classes, including numbers, strings and lists, so why not `Pair` objects too? The ability to define operators differently for different classes is called ***operator overloading***. Operator overloading is an example of ***polymorphism***, a feature of object-oriented programming languages in which methods and operators respond differently to objects of different classes. For example, consider the following list of different objects:

```
>>> things = [42, 'eggs ', [1, 2, 3], 3.14]
```

If we multiply every item in this list by `2`, the multiplication operator will act differently for each item, appropriate to its class:

```
>>> for item in things:
        print(item * 2)
84
eggs eggs
[1, 2, 3, 1, 2, 3]
6.28
```

When the `+` operator is used, a *special method* named `__add__` is implicitly called (like how `__init__` is implicitly called by the constructor). In other words, an assignment statement like

```
name = first + last
```

is identical to

```
name = first.__add__(last)
```

The ability to define this special method for each class is what allows us to use the `+` operator in different ways on different objects. We can implement the `+` operator on `Pair` objects by simply changing the name of our `add` method to `__add__`:

```
    def __add__(self, pair2):
        """ (docstring omitted) """

        sumA = self._a + pair2._a
        sumB = self._b + pair2._b
        return Pair(sumA, sumB)
```

With this special method defined, we can carry out our previous example as follows:

```
duo1 = Pair(3, 8)
duo2 = Pair(4, 5)
sumPair = duo1 + duo2   # sumPair is assigned Pair(7, 13)
print(sumPair.get())    # prints "(7, 13)"
```

Reflection 12.18 *Incorporate the* `__add__` *method into your* `Pair` *class and experiment with adding* `Pair` *objects.*

Reflection 12.19 *The behavior of the* `-` *operator is similarly defined by the* `__sub__` *method. Modify your* `subtract` *method so that it is called when the* `-` *operator is used with* `Pair` *objects.*

Similarly, we can define the `*` and `/` operators to implement multiplication and division with `Pair` objects. The methods corresponding to these operators are named `__mul__` and `__truediv__`, respectively. (Recall that `/` is called true division in Python while `//` is called floor division. The `//` operator is defined by the `__floordiv__` method.) Defining multiplication of a `Pair` object by a scalar quantity is similar to the `scale` method, but we return a new `Pair` instead.

```
    def __mul__(self, scalar):
        """Return a new Pair representing self multiplied by scalar."""

        return Pair(self._a * scalar, self._b * scalar)
```

Reflection 12.20 *How is the* `__mul__` *method different from* `scale`*? What does each return?*

With this new method, we can easily scale pairs of numbers in one statement:

```
bets = Pair(150, 100)
double = bets * 2      # double is assigned Pair(300, 200)
```

This assignment statement is equivalent to

```
double = bets.__mul__(2)
```

(but we would never call it that way).

Reflection 12.21 *Why would typing* `double = 2 * bets` *instead give an error?*

Reflection 12.22 *Using the* `__mul__` *operator as a template, write the* `__truediv__` *method to divide a* `Pair` *object by a scalar value.*

Applying the new addition and (true) division operators to our `centroid` function makes the code much more elegant!

```
def centroid(points):
    """Compute the centroid of a list of Pair objects."""

    n = len(points)
    if n == 0:
        return None

    sumPair = Pair()
    for point in points:
        sumPair = sumPair + point
    return sumPair / n
```

You may have already noticed that printing a `Pair` object is not very helpful:

```
>>> myPair = Pair(5, 7)
>>> print(myPair)
<__main__.Pair object at 0x1063551d0>
```

This is a default string representation of an object; it tells us that `myPair` is a `Pair` object in the `__main__` namespace, located in memory at address `1063551d0` (in hexadecimal notation). To override the default printing behavior for a `Pair`, we need to define another special method named `__str__`. The `__str__` method is called implicitly whenever we call the `str` function on an object. In other words, calling

```
str(myPair)
```

is identical to calling

```
myPair.__str__()
```

Since the `print` function implicitly calls `str` on an object, defining the `__str__` method also dictates how `print` behaves. The following `__str__` method for the `Pair` class returns a string representing the pair in parentheses (like a tuple).

```
    def __str__(self):
        """Return an '(a, b)' string representation of self."""

        return '(' + str(self._a) + ', ' + str(self._b) + ')'
```

With this method added to the class, if we want to include a string representation of a `Pair` object in a larger string, we can do something like this:

```
print('The current values are ' + str(myPair) + '.')
```

Also, just calling `print(myPair)` will now print `(5, 7)`.

Reflection 12.23 *Add the* `__str__` *method to your* `Pair` *class. Then print some of the* `Pair` *objects in your* `main` *function.*

Comparison operators

We can also overload the comparison operators `==`, `<`, `<=`, etc. using the following special methods.

Operator	==	!=	<	<=	>	>=
Method	`__eq__`	`__ne__`	`__lt__`	`__le__`	`__gt__`	`__ge__`

We will start by defining how the `==` operator behaves by defining the special method `__eq__`. It is natural to say that two pairs are equal if their corresponding values are equal, as the following method implements.

```
    def __eq__(self, pair2):
        """Return whether self and pair2 have the same values."""

        return (self._a == pair2._a) and (self._b == pair2._b)
```

Let's also override the < operator. If `duo1` and `duo2` are two `Pair` objects, then `duo1 < duo2` should return `True` if `duo1._a < duo2._a`, or if `duo1._a == duo2._a` and `duo1._b < duo2._b`. Otherwise, it should return `False`.

```
    def __lt__(self, pair2):
        """Return whether self < pair2."""

        return (self._a < pair2._a) or \
               ((self._a == pair2._a) and (self._b < pair2._b))
```

Suppose we store the number of wins and ties in a `Pair` object for each of three teams. If a team is ranked higher when it has more wins, and the number of ties is used to rank teams with the same number of wins, then the comparison operators we defined can be used to decide rankings.

```
wins1 = Pair(6, 2)     # 6 wins, 2 ties
wins2 = Pair(6, 4)     # 6 wins, 4 ties
wins3 = Pair(6, 2)     # 6 wins, 2 ties

print(wins1 < wins2)   # prints "True"
print(wins2 < wins3)   # prints "False"
print(wins1 == wins3)  # prints "True"
```

With the `__eq__` and `__lt__` methods defined, Python will automatically deduce the outcomes of the other four comparison operators. However, we will still leave their implementations to you as practice exercises.

Reflection 12.24 *Add these two new methods to your* `Pair` *class. Experiment with some comparisons, including those we did not implement, in your* `main` *function.*

Indexing

When an element in a string, list, tuple, or dictionary is accessed with indexing, a special method named `__getitem__` is implicitly called. For example, if `maxPrices` and `minPrices` are lists, then

```
priceRange = maxPrices[0] - minPrices[0]
```

is equivalent to

```
priceRange = maxPrices.__getitem__(0) - minPrices.__getitem__(0)
```

Similarly, when we use indexing to change the value of an element in a sequence, a method named `__setitem__` is implicitly called. For example,

```
temperatures[1] = 18.9
```

is equivalent to

```
temperatures.__setitem__(1, 18.9)
```

In the `Pair` class, we can use indexing with `__getitem__` as an alternative to the `getFirst` and `getSecond` methods to access the individual values in a `Pair` object.

```
    def __getitem__(self, index):
        """Return the first (index 0) or second (index 1) value in self.
           For other index values, return None."""

        if index == 0:
            return self._a
        if index == 1:
            return self._b
        return None
```

The `__getitem__` method returns the value of `self._a` or `self._b` if `index` is 0 or 1, respectively. If index is anything else, it returns `None`.

Reflection 12.25 *Is this behavior consistent with what happens when you use an erroneous index with a list?*

When we use an erroneous index with an object from one of the built-in classes, we get a `IndexError`. We will look at how to implement this alternative behavior in Sections 12.4 and 12.5.

As an example, suppose we have a `Pair` object defined as follows:

```
counts = Pair(12, 15)
```

With the new `__getitem__` method, we can retrieve the individual values in `counts` with

```
first = counts[0]
second = counts[1]
```

as these statements are equivalent to

```
first = counts.__getitem__(0)
second = counts.__getitem__(1)
```

Next, we can implement the `__setitem__` method as follows.

```
    def __setitem__(self, index, value):
        """Set the first (index 0) or second (index 1) item to value."""

        if index == 0:
            self._a = value
        elif index == 1:
            self._b = value
```

The `__setitem__` method assigns `self._a` or `self._b` to the given `value` if `index` is 0 or 1, respectively.

Reflection 12.26 *What does the* `__setitem__` *method do if* `index` *is not 0 or 1?*

With the new `__setitem__` method, we can assign a new value to `counts` with

```
counts[0] = 14
counts[1] = 16
print(counts)    # prints "(14, 16)"
```

With these indexing methods defined, we can now also use indexing within other methods, as convenient. For example, we can use indexing in the `__add__` method to get the individual values and in the `set` method to assign new values.

```
    def __add__(self, pair2):
        """ (docstring omitted) """

        sumA = self[0] + pair2[0]
        sumB = self[1] + pair2[1]
        return Pair(sumA, sumB)

    def set(self, a, b):
        """ (docstring omitted) """

        self[0] = a
        self[1] = b
```

Reflection 12.27 *Add the two indexing methods to your* `Pair` *class. Then modify the* `__lt__` *method so that it uses indexing to access values of* `self._a` *and* `self._b` *instead.*

You can find a summary of these and other special methods in Appendix A.9.

Exercises

12.2.1* Add a method to the `Pair` class named `round` that rounds the two values to the nearest integers.

12.2.2* Suppose you are tallying the votes in an election between two candidates. Write a program that repeatedly prompts for additional votes for both candidates, stores these votes in a `Pair` object, and then adds this `Pair` object to a running sum of votes, also stored in `Pair` object. For example, your program output may look like this:

```
Enter votes (q to quit): 1 2
Enter votes (q to quit): 2 4
Enter votes (q to quit): q

Candidate 1: 3 votes
Candidate 2: 6 votes
```

12.2.3* Suppose you are writing code for a runner's watch that keeps track of a list of split times and total elapsed times. While the timer is running, and the split button is pressed, the time elapsed since the last split is recorded in a `Pair` object along with the total elapsed time so far. For example, if the split button were pressed at 65, 67, and 62 second intervals, the list of (split, elapsed) pairs would be `[(65, 65), (67, 132), (62, 194)]` (where a tuple represents a `Pair` object). Write a function that is meant to be called when the split button is pressed to update this list of `Pair` objects. Your function should take two parameters: the current list of `Pair` objects and the current split time.

12.2.4. A data logging program for a jetliner periodically records the time along with the current altitude in a `Pair` object. Write a function that takes such a list of `Pair` objects as a parameter and plots the data using `matplotlib`.

12.2.5. Write a function that returns the distance between two two-dimensional points, each represented as a `Pair` object.

12.2.6. Write a function that returns the average distance between a list of points, each represented by a `Pair` object, and a given site, also represented as a `Pair` object.

12.2.7. The file `africa.txt`, available on the book website, contains (longitude, latitude) locations for cities on the African continent. The following program reads this file into a list of `Pair` objects, find the closest and farthest pairs of points in the list, and then plot all of the points using turtle graphics, coloring the closest pair blue and farthest pair red. Finish this program by adding a method named `draw(self, tortoise, color)` to the `Pair` class that plots a `Pair` object as an (x,y) point, and writing the functions named `closestPairs` and `farthestPairs`.

```
import turtle

class Pair:
    FILL IN THE CLASS HERE FROM THE TEXT

    def draw(self, tortoise, color):
        pass

def closestPairs(points):
    pass

def farthestPairs(points):
    pass

def main():
    points = []
    inputFile = open('africa.txt', 'r', encoding = 'utf-8')
    for line in inputFile:
        values = line.split()
        longitude = float(values[0])
        latitude = float(values[1])
        p = Pair(longitude, latitude)
        points.append(p)

    cpoint1, cpoint2 = closestPairs(points)
    fpoint1, fpoint2 = farthestPairs(points)

    george = turtle.Turtle()
    screen = george.getscreen()
    screen.setworldcoordinates(-37, -23, 37, 58)
    george.hideturtle()
    george.speed(0)
    screen.tracer(10)
    for point in points:
        point.draw(george, 'black')
    cpoint1.draw(george, 'blue')
    cpoint2.draw(george, 'blue')
    fpoint1.draw(george, 'red')
    fpoint2.draw(george, 'red')
    screen.update()
    screen.exitonclick()

main()
```

12.2.8. Rewrite the `Pair` class so that it stores its two values in a two-element list instead. The way in which the class' methods are called should remain exactly the same. In other words, the way someone uses the class (the ADT specification) must remain the same even though the implementation changes.

12.2.9* Implement alternative `__mul__` and `__truediv__` methods for the `Pair` class that multiply two `Pair` objects. The product of two `Pair` objects `pair1` and `pair2` is a `Pair` object in which the first value is the product of the first values of `pair1` and `pair2`, and the second value is the product of the second values of `pair1` and `pair2`. Division is defined similarly.

12.2.10. Implement the remaining four comparison operators (`!=`, `<=`, `>`, `>=`) for the `Pair` class.

12.2.11. Rewrite your `linearRegression` function from Exercise 7.6.1 so that it takes a list of `Pair` objects as a parameter.

12.2.12. Add a `__str__` method to the president class that you wrote in Exercise 12.1.10. The method should return a string containing the president's name and political party, for example, `'Kennedy (D)'`. Also, write a function that, given a list of president objects and a state abbreviation, prints the presidents in this list (indirectly using the new `__str__` method) that are from that state.

12.2.13. Add a `__lt__` method to the president class that you wrote in Exercise 12.1.10. The method should base its results on a comparison of the presidents' ages.

12.2.14. Add a `__str__` method to the `Senator` class from Exercise 12.1.12 that prints the name of the senator followed by their party, for example, `'Brown, Sherrod (D)'`. Also modify your program from part (b) so that it uses the new `__str__` method.

12.2.15. Rewrite the distance function from Exercise 12.2.5 so that it uses indexing to get the first and second values from each pair.

12.2.16. Write a class that represents a rational number (i.e., a number that can be represented as a fraction). The constructor for your class should take a numerator and denominator as parameters. In addition, implement the following methods for your class:

- arithmetic: `__add__`, `__sub__`, `__mul__`, `__truediv__`
- comparison: `__lt__`, `__eq__`, `__le__`
- `__str__`

When you are done, you should be able to perform calculations like the following:

```
a = Rational(3, 2)    # 3/2
b = Rational(1, 3)    # 1/3
total = a + b
print(total)          # should print 11/6
print(a < b)          # should print False
```

*12.3 A FLOCKING SIMULATION

This section is available on the book website.

*12.4 A STACK ADT

This section is available on the book website.

*12.5 A DICTIONARY ADT

This section is available on the book website.

12.6 SUMMARY AND FURTHER DISCOVERY

When we design an algorithm using an object-oriented approach, we begin by identifying the main objects in our problem, and then define abstract data types for them. When we design a new ADT, we need to identify the data that the ADT will contain and the operations that will be allowed on that data. These operations are generally organized into three categories: *constructors*, *accessors*, and *mutators*.

In an *object-oriented programming language* like Python, abstract data types are implemented as *classes*. A Python class contains a set of functions called *methods* and a set of *instance variables* whose names are preceded by `self` within the class. The name `self` always refers to the object on which a method is called. Bundling instance variables and methods together in a class, and restricting public access to the instance variables, is known as *encapsulation*, a key feature of object-oriented programming.

A class can also define the meaning of several *special methods* that dictate how operators and built-in functions behave on the class. These special methods partially implement another feature of object-oriented languages called *polymorphism*, which refers to the ability of a programming language to do different things when the same method is called on objects from different classes.

The manner in which a class implements the specification given by the abstract data type is called a *data structure*. There may be many different data structures that one can use to implement a particular abstract data type. For example, the Pair ADT from the beginning of this chapter may be implemented with two individual variables, a list of length two, a two-element tuple, or a dictionary with two entries.

To illustrate how classes are used in larger programs, we designed an *agent-based simulation* that simulates a viral epidemic in the first section and then a more complex simulation of flocking birds or schooling fish later on. These simulations consist of two main classes that interact with each other: an agent class and a class for the world that the agents inhabit. Agent-based simulations can be used in a variety of disciplines including sociology, biology, economics, and the physical sciences.

Finally, we designed two *collection* classes, a Stack and a Dictionary, to demonstrate how more complex classes can be implemented in Python.

Notes for further discovery

This chapter's epigraph is from one of the first papers written by Barbara Liskov, in 1974 [35]. In 1968, Dr. Liskov was one of the first women to earn a Ph.D. in computer science in the United States. She has taught computer science at MIT since 1972, and was honored with the Turing Award in 2008.

The boids model was created by Craig Reynolds [54]. For more information on agent-based simulations, we suggest looking at *The Computational Beauty of Nature* by Gary Flake [17], *Think Complexity* by Allen Downey [13], *Agent-based Models* by Nigel Gilbert [19], and *Introduction to Computational Science* by Angela Shiflet and George Shiflet [61].

*12.7 PROJECTS

This section is available on the book website.