> *The architects themselves came in to explain the advantages of both designs.*

Class Design

In the previous sections, we have used existing classes (e.g. str and list) that enable us to be able to write functions and programs that do what we wish. In this section, we will begin to design our own classes. Being able to write our own classes is a useful tool as not everything is in a format that we like or will find easy to use for a particular program that we may wish to write. We will start with the design of a simple ADT for a 2D point (an x and y coordinate).

2D Point Class

For our first example will we write the class definition for a 2D point. This class will require a *constructor* (to be able to create instances of the class), a couple of *accessors* that get the x, y coordinate of the point and a *mutator* to move the point by a certain distance. The class definition is as follows. Below, we will discuss this definition in detail.

```
import math

class Point(object) :
    """A 2D point ADT using Cartesian coordinates."""

    def __init__(self, x, y) :
        """Construct a point object based on (x, y) coordinates.

        Parameters:
            x (float): x coordinate in a 2D cartesian grid.
            y (float): y coordinate in a 2D cartesian grid.
        """
        self._x = x
        self._y = y

    def x(self) :
        """(float) Return the x coordinate of the point."""
        return self._x

    def y(self) :
        """(float) Return the y coordinate of the point."""
        return self._y

    def move(self, dx, dy) :
        """Move the point by (dx, dy).

        Parameters:
            dx (float): Amount to move in the x direction.
            dy (float): Amount to move in the y direction.
        """
        self._x += dx
        self._y += dy
```

Class definitions start with the keyword class, followed by the class name, and `(object):`. Following this, and indented, is what looks like function definitions. These are the **method definitions** for the class. Notice the first argument of each method is `self`: `self` is a reference to the object itself. This argument is needed so that the method can access and modify

components of the object. Class names, like function names, follow a naming convention. This convention is that the first letter of each word in the class name is a capital letter. Methods follow the same naming convention as functions.

Class Definition Syntax

```
class ClassName(object) :
    """Comment"""

    def method_1(self, [args]) :
        method_1_body

    def method_2(self, [args]) :
        method_2_body

    ...
```

Semantics

Creates a class called `ClassName` to represent the ADT specified. The methods of the class are `method_1`, `method_2`, and so on. Each method must have a parameter `self` to represent the instance that is performing the method, optionally followed by any other arguments the method requires.

Method calls of the form `instance.method(arg1, arg2, ...)` will execute the body of the definition of `method` with the arguments `instance, arg1, arg2, ...`. Note that the value of the `self` parameter is `instance`.

Setting up a Point

Earlier in the course, we have used the `dir` function to list the methods of an object, and saw that many of the methods had double underscores at the start and end of their names. These methods each have a special meaning in Python, which allow the object to work with existing Python syntax, such as arithmetic, slicing, and built-in functions.

The first method that the Point class has is the `__init__` method. This is the **constructor** method of the class, which is executed when the object is created. `__init__` takes as arguments any data that is to be required to make the instance of the class and creates the **instance variables** and any data structures required for the instance to function. In this case `__init__` takes in the x, y coordinate of the point and creates two instance variables, `self._x` and `self._y`. Note the `self.` at the start of the variable names. `self` is required to access any variables and methods of the instance, as it is the reference to the object instance. `self._x` and `self._y` are given the values of `x` and `y` respectively from the inputs into the constructor (`__init__` method) of the Point class.

The underscores on the instance variables also have an informal meaning. In object-oriented programming it is often useful to have **private** variables and methods. This is data and methods that are not meant to be accessed from outside the class except perhaps via a non-private method. Python does not have a way of setting variables and methods private. Python instead uses underscores to 'hide' variables and methods. ***Note***: This is simply a naming convention used by programmers to signify to readers of the code that this variable or method is meant to be private. It is possible to access these variables and methods directly if the number of underscores is know, though it is not recommended in case the class definition changes.

The next two methods (the x and y methods) are our **accessors** — they provide us with an interface to access the class instance variables from outside the class definition. These methods simply return the corresponding coordinate (either x or y). The method move is a **mutator** — it modifies the data stored in the object. Other than `self`, `move` also has the arguments `dx` and `dy`. These are added to the current `self._x` and `self._x`, respectively, to 'move' our point along a certain vector to a new location.

Now we can save our code as point1.py and run a few tests to see it in action.

```
>>> p = Point(2, -5)
>>> p
<__main__.Point object at 0x011D4710>
>>> str(p)
'<__main__.Point object at 0x011D4710>'
>>> type(p)
<class '__main__.Point'>
>>> p.x()
2
>>> p.y()
-5
>>> p.move(-3, 9)
>>> p.x()
-1
>>> p.y()
4
```

The first line creates an instance, `p`, of our new Point class. When we create a `Point` instance, we are **instantiating** the class. The example shows a print out of how Python represents our Point class, while the one after it shows Pythons string version of our Point class. The following example shows the type of our Point class. We then go and call the `x` and `y` methods to view the current state of the class instance. We then move our Point, `p`, and then have a look at the new state of the class instance.

String Representations

The second and third examples above showed how Python represents our Point class. The Python interpreter is using the default methods for `str` and `repr` for the string representations of our class. These are not particularly nice or useful representations of this class as it does not tell us much about the instance state. We can make our class have a better representation by writing our own `__str__` and `__repr__` methods into our class definition. These two method names are used to define the behaviour of the built-in `str` and `repr` functions respectively. The following method definitions will provide good `__str__` and `__repr__` representations of our Point.

```
    def __str__(self) :
        """The 'informal' string representation of the point."""
        return '({0}, {1})'.format(self._x, self._y)

    def __repr__(self) :
        """The 'official' string representation of the point."""
        return 'Point({0}, {1})'.format(self._x, self._y)
```

The __str__ and __repr__ methods both use a similar formatted string to produce a string that is a nice representation of our Point class. These methods are used when the functions str and repr, respectively, are used on our Point class. __repr__, ideally, should represent all the data

important to the object's state and to be able to recreate the object with that same state. If possible we should also make it so that if the interpreter read the repr string back in and evaluated it, it would construct a copy of the object.

After adding the above code to the class definition our class we can now save our point2.py code and test our two new methods.

```
>>> p = Point(-1, 4)
>>> p
Point(-1, 4)
>>> str(p)
'(-1, 4)'
>>> repr(p)
'Point(-1, 4)'
```

The first example shows what Python now returns if we simply ask for `p`. When something is evaluated the interpreter uses the `__repr__` method of the class as is shown in the third example. The second example shows the string representation of our Point class. Notice that the repr string looks just like the line created to make the Point instance originally.

Arithmetic on Points

```
>>> Point(1, 3) == Point(1, 3)
False
>>> p1 = Point(2, 3)
>>> p2 = p1
>>> p1 == p2
True
>>> p3 = Point(2, 3)
>>> p1 == p3
False
```

The above examples show that if we create Points with the same parameters they are not equal, even though the share the same state. This is clearly shown in the first and last example. These examples return False as, even though both objects have the same state, they are different instances of the class. The second example returns True as there are two variables with the same instance of Point as their value. The reason the interpreter behaves this way is because the interpreter is using the default test for equality, that objects are equal if they are the same instance.

We would like to define how equality should work on points. The method `__eq__` is used to define the behaviour of the `==` test. It would also be useful if we were able to add two Points together. The `__add__` method is used to define addition of objects using the `+` operator. The following code contains the two method definitions for `__add__` and `__eq__` which will give us the functionality we want when the `+` or `==` operators are used.

```
    def __add__(self, other) :
        """Return a new Point after adding this point to 'other'.

        Perform vector addition of the points considered as vectors.
        point1 + point2 -> Point

        Parameters:
            other (Point): Other point to be added to this point.

        Return:
            Point: New point object at position of 'self' + 'other'.
        """
        return Point(self._x + other.x(), self._y + other.y())

    def __eq__(self, other) :
        """Return True iff 'self' and 'other' have the same x and y coords.

        point1 == point2 -> bool

        Parameters:
            other (Point): Other point to be compared to this point.

        Return:
            bool: True if 'self' and 'other' have the same x and y coords.
                  False otherwise.
        """
        return self._x == other.x() and self._y == other.y()
```

The __add__ method adds the two Points together using vector addition. Then creates a new Point and returns it. The __eq__ method returns `True` if the points have the same x, y coordinates, and `False` otherwise.

Here are some examples of these last two methods after adding them to the class definition and saving the point3.py file.

```
>>> p1 = Point(1, 5)
>>> p2 = Point(-2, 9)
>>> p3 = p1 + p2
>>> p3
Point(-1, 14)
>>> p4 = Point(-1, 14)
>>> p3 == p4
True
>>> p1 == p2
False
>>> p1 += p2
>>> p1 == p4
True
```

First we create 2 instances of the Point class. Then we create a third by adding the first 2 together. After creating a fourth instance of our Point class we do a couple of tests for equality. In the last two examples we perform a `+=` and then another test to demonstrate that `p1` now equals `p4`.

Special Methods

The Python interpreter recognises many special method names to allow classes to use built-in Python syntax. Each of these names begins and ends with two underscores. The names Python recognises include:

`__init__(self, [arg1, arg2, ...])`

Constructor, executed when an instance of the class is created.

`__str__(self)`

Must return a string giving an "informal" description of the object. Executed when `str(x)` or `print(x)` are called.

`__repr__(self)`

Must return a string giving a "formal", unambiguous description of the object. Executed when `repr(x)` is called, or when `>>> x` is executed at the prompt. This string representation is often useful in debugging.

`__add__(self, other)`
`__sub__(self, other)`
`__mul__(self, other)`
`__div__(self, other)`

Definition of addition, subtraction, multiplication and division. Equivalent to `self + other`, `self - other`, `self * other`, and `self / other` respectively.

`__lt__(self, other)`
`__le__(self, other)`
`__eq__(self, other)`
`__ne__(self, other)`
`__gt__(self, other)`
`__ge__(self, other)`

Definitions of comparison operators `<, <=, ==, !=, >, >=` respectively. For example, `x <= y` executes the method call `__le__(x, y)`.

We now consider extending this ADT by providing accessor methods for getting the polar coordinates of the point. The polar coordinates specify a point by angle and distance rather than x and y coordinates. The new methods are given below.

```
    def r(self) :
        """(float) Return the distance of the point from the centre of the
        coordinate system (0, 0).
        """
        return math.sqrt(self._x**2 + self._y**2)

    def theta(self) :
        """(float) Return the angle, in radians, from the x-axis of the point."""
        return math.atan2(self._y, self._x)
```

The r method uses mathematics we know to calculate the radial position of the Point from the centre of our coordinates. The theta method uses the math libraries `atan2` method to find the angle from the x-axis. `atan2` is more accurate than `atan` as `atan2` returns a correct angle no matter what quadrant the point is in, whereas `atan` returns between pi/2 and -pi/2 radians.

After adding these methods into the class definition and saving, the file should now look like point.py. We can now look at a few examples of our last couple of methods.

```
>>> p = Point(3, 4)
>>> p.r()
5.0
>>> p.theta()
0.9272952180016122
```

The Abstraction Barrier

Imagine we now write some graphics program that uses this ADT. Later we come back and reconsider the implementation of the ADT and decide to use polar coordinates rather than x, y coordinates for the internal representation of the data, like the point_rt.py file here. We make sure that the constructor and the method interfaces behave in the same way as before (have the same semantics). Now we go back to our graphics program we wrote. **Do we need to change anything? No!** Because we did not change the interface, we do not need to change anything in our graphics program. This is the key point about ADTs — we have completely separated the implementation from the use of the ADT via a well-defined interface. **We respect the abstraction barrier!**

Note: if our graphics program directly accessed the x and y coordinates instead of using the interface then we would be in trouble if we changed over to polar coordinates — we would have to rethink all our uses of Point objects! This is why we use the "private variable" naming convention to signal that the `_x` and `_y` values should not be accessed.

A Ball Class

We now design another class that is similar to the one above in some ways, but we would probably not think of it as an ADT because it has more 'behaviour'. What we want to do here is to model a ball and its motion on something like a billiard table without pockets.

To begin with, we look at what assumptions we will make. Firstly, to simplify the physics, we assume no friction and no loss of energy when the ball bounces off the table edge. Secondly, we assume a given radius of the ball and table dimensions. Lastly, we assume a given positive time step and that the time step is small enough to reasonably approximate the ball movement. For a first pass at this problem, we will also use the following **global constants**: `TOP`, `LEFT`, `BOTTOM`, `RIGHT` and `TIMESTEP` to describe the edges of the table and the time step. These, to Python, are variables but as we are not changing them in our code, they are called constants. It is naming convention to use all uppercase to indicate constants. We also assume all the balls have the same radius.

Next, we need to determine what is necessary to describe the **state** of the ball. We need to know its position, speed and direction. Finally, we need to know what methods we will need — in other words, what the ball interface will look like. We will need accessors to get the position, speed and direction and a mutator that modifies the balls state based on the given time step. We also add a test to determine if this ball is touching another ball. To do this we require some simple trigonometry and physics and so we will import the math module.

We will start with the class constructor and accessors and repr along with some of the base code as follows.

```
import math

TOP = 0.0
LEFT = 0.0
BOTTOM = 2.0
RIGHT = 4.0
TIMESTEP = 0.1

class Ball(object) :
    """A class for simulating the movement of a ball on a billiard table.

    Class Invariant:
        0 <= _direction <= 2*pi
        and
        LEFT + radius <= _x <= RIGHT - radius
        and
        TOP + radius <= _y <= BOTTOM - radius
        and
        0 <= _speed
    """

    radius = 0.1

    def __init__(self, x, y, speed, direction) :
        """Initialise a ball object with position, speed and direction.

        Parameters:
            x (float): x coordinate starting position of Ball.
            y (float): y coordinate starting position of Ball.
            speed (float): Speed at which Ball is moving.
            direction (float): Direction in which Ball is moving.

        Preconditions:
            The supplied values satisfy the class invariant.
        """
        self._x = x
        self._y = y
        self._speed = speed
        self._direction = direction

    def get_centre_x(self) :
        """(float) Return the x coordinate of the Ball's centre."""
        return self._x

    def get_centre_y(self) :
        """(float) Return the y coordinate of the Ball's centre."""
        return self._y

    def get_speed(self) :
        """(float) Return the speed of the Ball."""
        return self._speed

    def get_dir(self) :
        """(float) Return the direction in which the ball is travelling."""
        return self._direction
```

```
    def __repr__(self) :
        """Ball's string representation."""
        return 'Ball({0:.2f}, {1:.2f}, {2:.2f}, {3:.2f})'.format(
                self._x, self._y, self._speed, self._direction)
```

Firstly, in the comments for the class itself we have included a **class invariant**. This is similar to the loop invariant we briefly discussed in week 5. The idea is that the class invariant is a property that should be true over the lifetime of each object of the class. In other words, it should be true when the object is first created and after each method is called. This is typically a formula that interrelates the instance variables. (To shorten the formula we have omitted the `self.` from the instance variables.) Even in a simple class like this, the class invariant can be a big help when it comes to writing methods. In particular, for the step method we can assume the class invariant is true when the method is called, and given that, we need to guarantee the class invariant is true at the end of the method.

The next part of the class is the assignment to the `radius`. This is a **class variable**. Class variables are variables that are common to *all* instances of the class — all instances of the class share this variable and if any instance changes this variable all instances 'will see the change'. As an example if we execute `self.radius = 0.2` than all ball instances will now have that radius. Since all the balls have the same radius, we make it a class variable.

The constructor (the __init__ method) initialises the instance variables. There are then the four accessor methods which return the values of the instance variables. This is followed by the `__rer__` method so that we can print our Ball instances out nicely.

After we save the code in ball1.py we can test with a few examples.

```
>>> b = Ball(0, 2, 3, 1)
>>> b
Ball(0.00, 2.00, 3.00, 1.00)
>>> b.get_centre_x()
0
>>> b.get_centre_y()
2
>>> b.get_dir()
1
>>> b.get_speed()
3
>>> b.radius
0.1
```

This is not particularly useful so let's look at writing the step method that calculates position of the ball in the next `TIMESTEP` and moves the ball to that location. This method is going to require two other methods to enable the ball to bounce off the walls if it reaches one. The following is the code of the methods.

```
    def _reflect_vertically(self) :
        """Change the direction as the ball bounces off a vertical edge."""
        self._direction = math.pi - self._direction
        if self._direction < 0 :
            self._direction += 2 * math.pi

    def _reflect_horizontally(self) :
        """Change the direction as the ball bounces off a horizontal edge."""
        self._direction = 2 * math.pi - self._direction
```

```
    def step(self) :
        """Advance time by TIMESTEP - moving the ball."""

        self._x += TIMESTEP * self._speed * math.cos(self._direction)
        self._y += TIMESTEP * self._speed * math.sin(self._direction)
        if self._x < LEFT + self.radius :
            self._x = 2 * (LEFT + self.radius) - self._x
            self._reflectVertically()
        elif self._x > RIGHT - self.radius :
            self._x = 2 * (RIGHT - self.radius) - self._x
            self._reflectVertically()

        if self._y  < TOP + self.radius :
            self._y = 2 * (TOP + self.radius) - self._y
            self._reflectHorizontally()
        elif self._y > BOTTOM - self.radius :
            self._y = 2 * (BOTTOM - self.radius) - self._y
            self._reflectHorizontally()
```

Notice the methods _reflect_vertically and _reflect_horizontally begin with underscores. As we do not want these two functions to be accessed outside the class, we flag them as private.

These two methods do exactly what they are named. _reflect_horizontally reflects the ball off any horizontal edge. This method simply takes the direction of the ball away from 2 pi, perfectly bouncing the ball off the wall at the same angle it hit the wall. _reflect_vertically is a little trickier as we need to make sure our class invariant is not false. To bounce off a vertical wall we simply take our direction away from pi. This is mathematically correct but it could make our class invariant false. For example, if the ball is travelling at pi + 0.1 radians and we do this bounce then our direction is now -0.1 radians. As this is a negative number we add 2 pi to it so that we get the positive angle (2 pi - 0.1 radians). This is the same angle as -0.1 radians and makes our class invariant true again.

The step method starts off updating the x, y coordinates of the ball by increasing the relevant coordinate by the `TIMESTEP` times the speed times the relevant component of the direction. Next, it needs to check if the ball has met a boundary of the table so that the ball does not break the class invariant by leaving the table. This is done, first by checking if it has left a vertical edge by seeing if the ball's x position is within radius distance of a vertical wall. If it is then the x position is shifted so that it is outside a radius distance of the wall and reflected using the `_reflect_vertically` method. A similar method is used for if the ball is on a horizontal edge.

> **Aside: Proving the Class Invariant**
> **Warning: This is not an easy proof!**
>
> We need to show that each method preserves the class invariant. The following is a proof of each method and how they preserve the class invariant.
>
> We start with the easier one — _reflect_horizontally. Let `d0` and `d1` be the initial and final values of `self._direction`.

```
We want to show that if
0 <= d0 <= 2*pi
then
0 <= d1 <= 2*pi
where
d1 == 2*pi - d0
(We use ==> for 'implies' below)

0 <= d0 <= 2*pi
==>
0 >= -d0 >= -2*pi           (multiplying by -1)
==>
2*pi >= 2*pi - d0 >= 0      (adding 2*pi)

QED
```

We now prove the property is true for _reflect_vertically. Here we let `d0` be the initial value of `self._direction`, `d1` be the value after the first assignment and `d2` be the final value. In this case there is an if statement involved and so we have to consider two cases: `d1 >= 0` and `d1 < 0`.

The first case.

```
We want to show that if
0 <= d0 <= 2*pi and d1 >= 0
then
0 <= d2 <= 2*pi
In this case the body of the if statement is not executed and so
d2 == d1 == pi - d0

0 <= d0 <= 2*pi
==>
0 >= -d0 >= -2*pi         (multiplying by -1)
==>
pi >= pi - d0 >= -pi      (adding pi)
==>
pi >= pi - d0 >= 0        (d1 >= 0 i.e. pi - d0 >= 0)

QED


The second case.

We want to show that if
0 <= d0 <= 2*pi and d1 < 0
then
0 <= d2 <= 2*pi
In this case the body of the if statement is executed and so
d2 == 2*pi + d1  and d1 == pi - d0 and so d2 == 3*pi - d0

d1 < 0
==>
pi - d0 < 0
==>
```

```
3*pi -d0 < 2*pi            (adding 2*pi)
and
d0 <= 2*pi
==>
-d0 >= -2*pi               (multiplying by -1)
==>
3*pi - d0 >= pi            (adding 3*pi)
and so
pi <= 3*pi - d0 <= 2*pi

QED
```

Now we look at the hardest part of the proof — that the ball stays on the table. The method has four if statements and below we will only consider the case when the first test is satisfied — the other cases follow in a similar manner. We let `x0` be the initial value of `self._x`, `x1 be the value` after the first assignment and `x2` be the final value. We also let `s` be `self._speed`, `d` be `self._direction` and `r` be `Ball.r`.

```
So we can assume
left+r <= x0 <= right-r
and
0 <= s*timestep < r
and
x1 < left + r    (the test in the first if statement is true)
and we want to show
left+r <= x2 <= right-r

We have
x1 == x0 + s*timestep*cos(d) and x2 == 2*(left+r) - x1

Now
x1 < left + r
==>
-x1 >= -left - r                    (multiplying by -1)
==>
2*(left+r) - x1 >= left+r           (adding 2*(left+r))
==>
x2 >= left+r
(one half of the required inequality)

We now need to show
2*(left+r) - x0 - s*timestep*cos(d) <= right-r

left+r <= x0
==>
left+r - x0 <= 0
==>
2*(left+r) - x0 + r <= left + 2*r                    (adding left + 2*r)
==>
2*(left+r) - x0 + s*timestamp <= left + 2*r          (r >= s*timestamp)
==>
2*(left+r) - x0 - s*timestamp*cos(d) <= left + 2*r   (1 >= -cos(d) and
s*timestamp >= 0)
==>
x2 <= left + 2*r
```

```
So provided left+2*r <= right - r  (i.e. right - left >= 3*r)
then the required property is true.
```

This means that, if we insist that the table is at least one and a half balls long and wide then the step method will maintain the class invariant.

The point of this exercise is to show that it is possible to prove useful properties about the class and therefore of any object of the class. Here we showed that, provided the global variables satisfy some reasonable constraints, any ball from the Ball class (that initially satisfies the class invariant) will stay on the table.

After adding the methods and saving the code ball2.py we can run a simple test.

```
>>> b = Ball(0.51, 0.51, 1.0, math.pi/4)
>>> for i in range(25) :
        b.step()
        print(b)

Ball(0.58, 0.58, 1.00, 0.79)
Ball(0.65, 0.65, 1.00, 0.79)
Ball(0.72, 0.72, 1.00, 0.79)
Ball(0.79, 0.79, 1.00, 0.79)
Ball(0.86, 0.86, 1.00, 0.79)
Ball(0.93, 0.93, 1.00, 0.79)
Ball(1.00, 1.00, 1.00, 0.79)
Ball(1.08, 1.08, 1.00, 0.79)
Ball(1.15, 1.15, 1.00, 0.79)
Ball(1.22, 1.22, 1.00, 0.79)
Ball(1.29, 1.29, 1.00, 0.79)
Ball(1.36, 1.36, 1.00, 0.79)
Ball(1.43, 1.43, 1.00, 0.79)
Ball(1.50, 1.50, 1.00, 0.79)
Ball(1.57, 1.57, 1.00, 0.79)
Ball(1.64, 1.64, 1.00, 0.79)
Ball(1.71, 1.71, 1.00, 0.79)
Ball(1.78, 1.78, 1.00, 0.79)
Ball(1.85, 1.85, 1.00, 0.79)
Ball(1.92, 1.88, 1.00, 5.50)
Ball(1.99, 1.81, 1.00, 5.50)
Ball(2.07, 1.73, 1.00, 5.50)
Ball(2.14, 1.66, 1.00, 5.50)
Ball(2.21, 1.59, 1.00, 5.50)
Ball(2.28, 1.52, 1.00, 5.50)
```

The last method we will define is a method to see if 2 balls are touching. This method will be useful when we have multiple Balls. Here is the method definition.

```
    def touching(self, other) :
        """(bool) Return True iff this Ball is touching other."""
        return (((self._x - other.get_centre_()) ** 2
                 + (self._y - other.get_centre_y()) ** 2)
                <= (2 * self.radius) ** 2)
```

This method gets the straight-line distance between the two balls and returns true if and only if (iff) the distance is less than or equal to the distance of two ball radii.

After adding this method to our class we can save our ball.py code.

Using the Ball Class

Where things get really interesting is when we create several instances of the class. Below is an example to show the power of object-oriented programming — once we have defined the class we can create as many instances as we want!

```
>>> balls = [Ball(1.0, 1.0, 1.0, 0),
             Ball(1.2, 1.2, 1.0, 1.0),
             Ball(1.4, 1.4, 1.0, 2.0)]
>>> balls
[Ball(1.00, 1.00, 1.00, 0.00), Ball(1.20, 1.20, 1.00, 1.00), Ball(1.40, 1.40,
1.00, 2.00)]
>>> for b in balls :
        b.step()

>>> balls
[Ball(1.10, 1.00, 1.00, 0.00), Ball(1.25, 1.28, 1.00, 1.00), Ball(1.36, 1.49,
1.00, 2.00)]

>>> def some_touch(balls) :
        for b1 in balls :
                for b2 in balls :
                        if b1 != b2 and b1.touching(b2) :
                                return True
        return False

>>> while not some_touch(balls) :
        for b in balls :
                b.step()

>>> balls
[Ball(1.20, 1.00, 1.00, 3.14), Ball(3.57, 1.49, 1.00, 4.14), Ball(1.13, 0.91,
1.00, 5.14)]
```

The next step would, of course, be to program the interaction between the balls. We could do this either by writing a collection of functions to manage the interaction of the balls and the motion or we could define a Table class (for example) which would contain a collection of balls and a step method for the table which would involve stepping each ball and defining how the balls bounce off each other.

Is defining a Table class worthwhile? It could be argued either way. If there was only ever going to be one table then it could be argued that creating a class would be overkill. On the other hand, collecting all the information and behaviour of the table into one place (a class) could be a good idea.

Summary

In this section we have introduced the idea of class design. Once we have defined a class we can take any number of instances of the class. This is a simple, but powerful form of **reuse**.

Things to consider when designing classes:

- What assumptions am I making?
- What data do I need to store?
    - Create instance variables to capture properties associated with individual objects.

    - Create class variables to capture shared properties
- Are the values of variables interrelated or constrained? — Add a class invariant to the class comments.
- What should the interface look like? — What are the 'public' methods?
- Name the class after the type of objects it produces.
- Name variables after their roles and make instance variables private.
- What information does the constructor need to create an object? Add parameters to the __init__ method and give them meaningful names.
- Name each method to suggest its role.
- Comment each method before writing any code!
- Don't 'over complicate' methods — methods should, where possible, perform just **one** task.
- What helper methods do I need? — make them private.
- For testing purposes write the __repr__ method.