Object-Oriented Programs

Moving from Procedural to Object-Oriented Programs

One of the key goals of object-oriented programming is software reuse. To achieve this we wrap methods and attributes in a class, and that makes it easier for other programs to use those classes. If we are just modelling data, maybe an array is all we need, and if we are just modelling behaviours, maybe some methods are all we need; but if we are modelling both data and behaviours, an object-oriented approach makes sense.

In this example we are calculating the perimeter of a shape. The code in **black** is the procedural version of the program, and the code in **red** is what we need to add in to make it object-oriented. We need to add in an `init` method for each class, and we also need to add in a method to take point values into the class, because with object-oriented design we prefer to encapsulate the values, and change them through a method.

```
# PROGRAM CalculatePerimeter:
import math

class Point:
    def _ _init_ _(self, x, y):
        self.x = x
        self.y = y
    # END init

    def distance(self, p2):
        return math.sqrt(
            (self.x - p2.x)**2 + (self.y - p2.y)**2)
    # END distance

# END class point

class Polygon:
    def _ _init_ _(self):
        self.vertices = []
    # END init

    def add_point(self, point):
        self.vertices.append((point))
    # END add_point

    def perimeter(self):
       perimeter = 0
       points = self.vertices + [self.vertices[0]]
       for i in range(len(self.vertices)):
           perimeter += points[i].distance(points[i+1])
       # ENDFOR
       return perimeter
    # END perimeter

# END class perimeter

# END.
```

The object-oriented code in **red**, doubles the length of the program.

Object-Oriented Programs

Running the Program

Below is how we would run the program procedurally, and how we would run it in an object-oriented way. As we can see in the procedural version we input the perimeter points directly, whereas in the object-oriented version we input them via a method. The object-oriented version certainly isn't more compact than the procedural version, but it is much clearer in terms of what is happening, and makes reuse far easier.

Procedural Version

```
>>> square =
[(1,1),(1,2),(2,2),(2,1)]

>>> perimeter(square)
```

Object-Oriented Version

```
>>> square = Polygon()
>>> square.add_point(Point(1,1))
>>> square.add_point(Point(1,2))
>>> square.add_point(Point(2,2))
>>> square.add_point(Point(2,1))
>>> print(square.perimeter())
```

Getters and Setters

As we mentioned, we prefer to access attributes through methods instead of accessing them directly.There are times when that applies to the internal code as well as to other classes. For example, if we want to assign a variable called `name` to a particular value, we would say something like `Colour.name = "Red"`, but, we could also write a method as follows, and to assign a value we would say `Colour.set_name("Red")`.

```
def set_name(self, name):
    self._name = name
# END set_name.
```

We can do the same for the command `print(Colour.name)` which we can change to become `print(Colour.get_name()` [We can do the same for a `return`]:

```
def get_name(self):
    return self._name
# END get_name.
```

The real benefit of getters and setters is that we can add conditions and checking into the getters and setters to make the code more robust and powerful:

```
def get_name(self):
    if self._name == "":
        return "There is a blank value"
    else:
        return self._name
    # ENDIF;
# END get_name.
```

If we have code that already does assignments and prints, we can force them to run as getters and setters using the property function as follows:

```
>>> name = property(_get_name, _set_name)
```

And now without having to change any code, the assignments, prints, and returns are upgraded to become getters (aka *accessor methods*) and setters (aka *mutator methods*).

Manager Objects

Manager Objects are like managers in offices, they tell other people what to do. The manager class and objects don't really do much activity themselves, and instead they call other methods, and pass messages between those methods.