Inheritance and Overriding Methods

Without getting too technical, the primary characteristics associated with object-oriented programming are

- inheritance;
- encapsulation; and
- polymorphism

Inheritance is a mechanism by which a class retains the state and behavior of another class. Encapsulation is about creating a public interface for your class and keeping the internal state sequestered. Polymorphism just means that a class is free to override a method from its base class and that the correct version of the method always gets called. In python, there is direct support for inheritance, encapsulation happens via naming conventions, and polymorphism happens by default—the most specific version of a method is always called, but one can use `super()` to refer to the super class.

Examples

Let's begin by defining a simple shape class as well as two classes that naturally extend from it—a rectangle class and a square class.

```
class Shape:

    def area(self):
        pass

class Rectangle(Shape):

    def __init__(self, width, height):
        self._width = width
        self._height = height

    def area():
        return self._width * self._height


class Square(Rectangle)

    def __init__(self, side):
        super().__init__(side, side)
```

Some notes:

- The `Shape` class is often called an *abstract class* because it does not define the `area` method. The purpose of `Shape` is to define an interface.
- The `Shape` class does not have an `__init__` method—the initialization method is not required, so if your class does not need to set up any internal state, you can safely disregard it.
- The `Rectangle` class inherits or extends the `Shape` class. We denote inheritance using the parenthesis in the class definition.

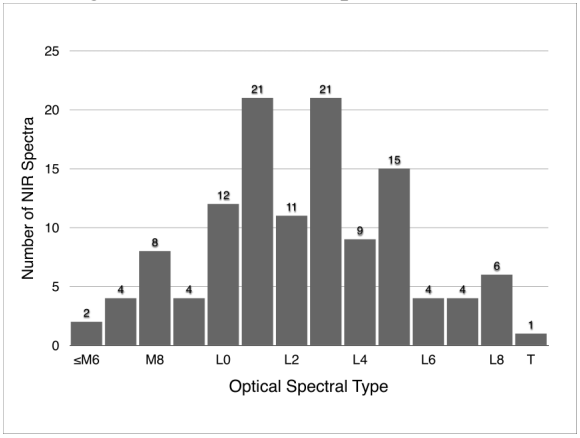
- The `Rectangle` class has two instance variables, `_width` and `_height`, which use the underscore convention to say to other programmers: *these instance variables are not meant to be accessed by you*. Other languages, like Java and C++ provide language support for this type of access control.
- The `Square` class extends `Rectangle`. Its initialization method uses the `super()` method to ask for the super class instance. This gives us access to all the methods defined in `Rectangle`, including its initialization.
- The `Square` class inherits the `area` method, so there's no need to redefine it.

```
>>> shape = Rectangle(10,20)
>>> shape.area()
200
>>> shape = Square(10)
>>> shape.area()
100
```

Let's begin by creating a chart class. A chart is not specific to any type of chart, so it just has a title. We are defining this chart for other classes to extend.

```
1  class Chart:
2
3      def __init__(self, title):
4          self._title = title
5
6      def title(self):
7          return self._title
8
9      def __str__(self):
10         return "{}".format(self._title)
```

A histogram is a chart that represents *counts* over a set of things, which we often call *bins*. Here is a histogram.



And here is a histogram class.

```
1  class Histogram(Chart):
2
3      def __init__(self, bins, title):
4          self._bins = bins
5          self._counts = [0]*len(self._bins)
6          super().__init__(title)
7
8      def _index(self, bin):
9          return self._bins.index(bin)
10
11     def add_to_bin(self, bin, count):
12         self._counts[self._index(bin)] += count
13
14     def count(self, bin):
15         return self._counts[self._index(bin)]
16
17     def __str__(self):
18         h = " ".join(["{}:{}".format(x,y) for (x,y) in zip(self._bins, self._counts)])
19         return "[{}] {}".format(super().__str__(), h)
```

```
>>> h = Histogram(["Intro", "Data Structures", "Algorithms", "Operating Systems"], "CS Course Enrollments")
>>> print(h)
[CS Course Enrollments] Intro:0 Data Structures:0 Algorithms:0 Operating Systems:0
>>> h.add_to_bin("Intro", 10)
>>> print(h)
[CS Course Enrollments] Intro:10 Data Structures:0 Algorithms:0 Operating Systems:0
>>> h.count("Intro")
10
>>> h.add_to_bin("Operating Systems", 30)
>>> print(h)
[CS Course Enrollments] Intro:10 Data Structures:0 Algorithms:0 Operating Systems:30
```

Static Methods

Static methods are functions associated with the class that do no rely on instance variables but still logically belong to the class. For example, our `Histogram` class might have a function `percentage` that takes a count and a total and returns the percentage—it's functionality that someone using a `Histogram` might want, but it doesn't rely on the instance variables. In Python, you use a *decorator* to indicate a static method with the `@staticmethod` syntax. Static methods don't take `self` parameters, so the methods look as follows:

```
1  class Histogram(Chart):
2
3      @staticmethod
4      def percentage(count, total):
5          return count / total
```