Classes

Classes define new types and their behavior. Using built-in types, we can create more complicated structures that with clearly defined interfaces. Creating new types gives us another tool for abstraction and the ability to write beautiful, readable, and often succinct code.

For example, in the past few labs we have been representing points as pairs of numbers. When we needed to find the distance between two points, we wrote a new function. But what if points were a type? And what if they knew how to compute their Euclidean distance to other points, or to add themselves to other points?

Here is an example of how we can use the Python class mechanism to define a new type.

```python
import math

class Point:

    origin = Point(0,0)

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def add(self, pt):
        return Point(self.x + pt.x, self.y + pt.y)

    def __add__(self, pt):
        return self.add(pt)

    def __repr__(self):
        return "Point({},{})".format(self.x, self.y)

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

    def __str__(self):
        return "({}.{})".format(self.x,self.y)
```

- Use the `class` keyword to define a class.
- Create new `Point` objects by invoking the class name using standard function notation. Here we'd create a point by invoking `Point(3,4)`. In this case, instantiating a `Point` means passing two numeric values, which comprise the *state* of the instance.
- The special method `__init__(self, ...)` allows us to specialize how an instance of `Point` is created. Notice that the parameter `self` is an implicit argument. This parameter will be bound to the newly created *instance*. You can use `self` to create attributes, bind, and retrieve attributes. The attributes created in the `__self__` method are usually called *instance variables*.
- Variables defined in the scope of the class, but not bound to the instance are called class variables.
- Function definitions inside the class definition that accept a `self` parameter are called *instance methods*.
- Function definitions inside the class definition that don't accept a `self` parameter are called *class methods*.

- The method `__add__` is called when when two points are *added* using the "+" syntax. In other words, "+" is syntactic sugar around the `__add__` method.
- The method `__repr__` is used to return a string representation of the object. This representation should uniquely identify the object. It is common to have this representation be the syntax for constructing the object in its current state.
- The method `__str__` is called when printing an object.
-

```
>>> p.distance(q)
1.0
>>> p+q
Point(0.5403023058681398,0.8414709848078965)
>>> p = Point(0,0)
>>> q = Point(math.cos(1),math.sin(1))
>>> q
Point(0.5403023058681398,0.8414709848078965)
>>> p.distance(q)
1.0
>>> r = Point(1,1)
>>> q + r
Point(1.5403023058681398,1.8414709848078965)
>>> p.origin == q.origin
True
>>> p.origin.x
0
>>> q.origin.x = 10
>>> p.origin.x
10
```

Here is a rectangle class that makes gratuitous use of the `Point` distance method when computing `width` and height.

```
class Rect:

  def __init__(self, pt0, pt1):
    self.pt0 = pt0
    self.pt1 = pt1

  def width(self):
    return self.pt0.distance(Point(self.pt1.x, self.pt0.y))

  def height(self):
    return self.pt0.distance(Point(self.pt0.x, self.pt1.y))

  def area(self):
    return self.width() * self.height()
```