18. Introduction to Classes

Topics:
- Class Definitions
- Constructors
- Example: The class Point
- Functions that work with Point Objects
- Defining methods
What a Simple Class Definition Looks Like

class Point(object):
    """
    Attributes:
        x: float, the x-coordinate of a point
        y: float, the y-coordinate of a point
    """
    def __init__(self,x,y):
        self.x = x
        self.y = y

A class can be used to “package” related data.
One Reason for classes: They Elevate the Level Thinking

>>> P = Point(2,1)
>>> Q = Point(6,4)
>>> d = P.Dist(Q)
>>> print d
5

Here, Dist is a method and P.Dist(Q) says “compute and return the distance from point P to point Q.”
One Reason for classes: They Elevate the Level Thinking

By having a `Point` class we can think at the “point level” instead of at the “xy level”
Classes and Types

Recall that a type is a set of values and operations that can be performed on those values.

The four basic "built-in" types:

   int, float, str, bool

Classes are a way to define new types.
Examples

By suitably defining a rectangle class, we could say something like

```python
if R1.intersect(R2):
    print 'Rectangles R1 and R2 intersect'
```
Examples

By suitably defining a polynomial class, we could perform operations like

\[ p = q + r \]

where \( q \) and \( r \) are polynomials that are added together to produce a polynomial \( p \)
How to Define a Class
class Point(object):
    
    Attributes:
    
        x: float, the x-coordinate of a point

        y: float, the y-coordinate of a point

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

A class provides a “blue print” for packaging data. The data is stored in the attributes.
A Point Class

class Point(object):
    """
    Attributes:
    x: float, the x-coordinate of a point
    y: float, the y-coordinate of a point
    """
    def __init__(self,x,y):
        self.x = x
        self.y = y

This special function, called a constructor, does the packaging.
A Point Class

class Point(object):
    
    Attributes:
    x: float, the x-coordinate of a point
    y: float, the y-coordinate of a point
    
    def __init__(self, x, y):
        self.x = x
        self.y = y

The name of this class is "Point"
The "__init__" Function

def __init__(self,x,y):
    """ Creates a Point object
    """
    PreC: x and y are floats
    """
    self.x = x
    self.y = y

That's a double underscore: __init__
The "\_\_init\_\_" Function

```python
def __init__(self, x, y):
    """ Creates a Point object
    PreC: x and y are floats
    """
    self.x = x
    self.y = y
```

"""self"" is always the first argument for any method defined in a class.
The “__init__” Function

def __init__(self, x, y):
    """ Creates a Point object
    PreC: x and y are floats
    """
    self.x = x
    self.y = y

    The attributes are assigned values.
Calling the Constructor
Creates an Object
Calling the Constructor

```python
>>> a = 3
>>> b = 4
>>> Q = Point(a, b)
```

The constructor's name is the name of the class.
Calling the Constructor

```python
>>> a = 3
>>> b = 4
>>> Q = Point(a, b)
```

This creates a `Point` object.
Calling the Constructor

The constructor returns a reference, in effect, the red arrow.
Objects: The Folder Metaphor

In the office, manila folders organize data.

Objects organize data.

A point object houses float variables $x$ and $y$, called the attributes, where $(x,y)$ is the point.
Objects: The Folder Metaphor

In the office manila folders organize data.

Objects organize data.

A color object might house an rgb list like [1,0,1] and a string that names it, i.e., ‘magenta’
Visualizing a Point Object

```python
>>> a = 3
>>> b = 4
>>> Q = Point(a, b)
```

Attributes are variables that live inside objects.

$x$ and $y$ are attributes.
Accessing an Attribute

The “Dot Notation” Again

Not a coincidence: modules are objects
Accessing Attributes

```python
>>> Q = Point(3,4)
```

```python
>>> print Q
( 3.000, 4.000)
```

```python
>>> Q.x = Q.x + 5
```

```python
>>> print Q
( 8.000, 4.000)
```

\texttt{Q.x} is a variable and can "show up" in all the usual places, i.e., in an assignment statement.
Accessing Attributes

```python
>>> Q = Point(3,4)
>>> print Q
( 3.000, 4.000)
>>> Q.x = Q.x + 5
>>> print Q
( 8.000, 4.000)
```

Seems that we can print an object!
The "__str__" function

```python
def __str__(self):
    return '(%6.3f,%6.3f)' % (self.x, self.y)
```

This "double underscore" function is part of the class definition.

Whenever a statement like

```
print P
```

is encountered, then P is "pretty printed" according to the format rules.
Two Examples

A function that returns a Point Object:

\[ \text{RandomPoint}(Lx, Rx, Ly, Ry) \]

A function that has input parameters that are Point objects:

\[ \text{Midpoint}(P, Q) \]
def RandomPoint(Lx,Rx,Ly,Ry):
    """ Returns a point that is randomly chosen from the square Lx<=x<=Rx, Ly<=y<=Ry. """

    PreC: Lx and Rx are floats with Lx<Rx
    Ly and Ry are floats with Ly<Ry

    """

    x = randu(Lx,Rx)
    y = randu(Ly,Ry)
    P = Point(x,y)
    return P

calling the constructor
def Midpoint(P1,P2):
    """ Returns a point that is the midpoint of a line segment that connects P1 and P2. """

    PreC: P1 and P2 are point objects.
    """

    xm = (P1.x + P2.x)/2.0
    ym = (P1.y + P2.y)/2.0
    Q = Point(xm,ym)
    return Q
def Midpoint(P1,P2):
    """ Returns a point that is the midpoint of the line segment that connects P1 and P2. """

    xm = (P1.x + P2.x)/2.0
    ym = (P1.y + P2.y)/2.0
    Q = Point(xm,ym)
    return Q
Methods

Methods are functions that are defined inside a class definition.

We have experience using them with strings

```
s.upper(), s.find(s1), s.count(s2),
s.append(s2), s.split(c), etc
```

and lists

```
L.append(x), L.extend(x), L.sort(), etc
```
Methods

Now we show how to implement them.

We will design a method for the `Point` class that can be used to compute the distance between two points.

It will be used like this:

```
delta = P.Dist(Q)
```

Note the dot notation syntax for method calls.
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

def Dist(self, other):
    """ Returns distance from self to other.  
    PreC: other is a point  
    ""
    dx = self.x - other.x
    dy = self.y - other.y
    d = sqrt(dx**2 + dy**2)
    return d

Assume proper importing from math class
Using the Dist Method

Let's create two point objects and compute the distance between them. This can be done two ways...

```python
>>> P = Point(3,4)
>>> Q = Point(6,8)
>>> deltaPQ = P.Dist(Q)
>>> deltaQP = Q.Dist(P)
>>> print deltaPQ,deltaQP
5.0 5.0
```

The usual "dot" notation for invoking a method
Method Implementation:
Syntax Concerns

class Point(object):
    
def Dist(self, other):
        """ Returns distance from self to other. 
        PreC: P is a point 
        """
        dx = self.x - other.x
        dy = self.y - other.y
        d = sqrt(dx**2 + dy**2)
        return d

Note the use of "self".
It is always the first argument of a method.
How to Think “Method”

class Point(object):
    
def Dist(self, other):
        """ Returns distance from self to other."
        PreC: P is a point
        """
        dx = self.x - other.x
        dy = self.y - other.y
        d = sqrt(dx**2 + dy**2)
        return d

Think like this: “We are going to apply the method dist to a pair of Point objects, self and other.”
Method Implementation: Syntax Concerns

class Point(object):
    :
    def Dist(self,other):
        """ Returns distance from self to other
        PreC: other is a point
        """
        dx = self.x - other.x
        dy = self.y - other.y
        d = sqrt(dx**2+dy**2)
        return d

Two Facts:
    Indentation is important.
    A class method is part of the class definition.
Visualizing a Method Call Using State Diagrams

Let’s see what happens when we execute the following:

\[
P = \text{Point}(3, 4) \\
Q = \text{Point}(6, 8) \\
D = P.\text{Dist}(Q)
\]
Visualizing a Method Call

- \( P = \text{Point}(3,4) \)
- \( Q = \text{Point}(6,8) \)
- \( D = P.\text{Dist}(Q) \)
P = Point(3,4)
Q = Point(6,8)
D = P.Dist(Q)
Visualizing a Method Call

\[ P = \text{Point}(3,4) \]
\[ Q = \text{Point}(6,8) \]
\[ D = P.\text{Dist}(Q) \]

\[ \begin{align*}
    dx &= \text{self.x} - \text{other.x} \\
    dy &= \text{self.y} - \text{other.y} \\
    d &= \sqrt{(dx^2 + dy^2)} \\
    \text{return } d
\end{align*} \]
Method: Dist

class Point(object):
    :
    def Dist(self, other):
        ""
        Returns distance from self to other.
        PreC: other is a point
        ""
        dx = self.x - other.x
        dy = self.y - other.y
        d = sqrt(dx**2+dy**2)
        return d

Think of **self** and **other** as input parameters.
Visualizing a Method Call

\[
P = \text{Point}(3, 4) \\
Q = \text{Point}(6, 8) \\
D = P.\text{Dist}(Q)
\]

\[
dx = \text{self}.x - \text{other}.x \\
dy = \text{self}.y - \text{other}.y \\
d = \sqrt{dx^2 + dy^2} \\
\text{return } d
\]

Control passes to the method Dist
P = Point(3,4)
Q = Point(6,8)
D = P.Dist(Q)

dx = self.x - other.x
dy = self.y - other.y
d = sqrt(dx**2 + dy**2)
return d
Visualizing a Method Call

\[ P = \text{Point}(3,4) \]
\[ Q = \text{Point}(6,8) \]
\[ D = P.\text{Dist}(Q) \]

\[
\begin{align*}
\text{dx} &= \text{self}.x - \text{other}.x \\
\text{dy} &= \text{self}.y - \text{other}.y \\
\text{d} &= \sqrt{(\text{dx}^2 + \text{dy}^2)} \\
\end{align*}
\]

return d

\[ d: 5 \hspace{1cm} \text{dx}: -3 \hspace{1cm} \text{dy}: -4 \]
Visualizing a Method Call

\[
P = \text{Point}(3,4) \\
Q = \text{Point}(6,8) \\
D = P.\text{Dist}(Q)
\]

\[
dx = \text{self}.x - \text{other}.x \\
dy = \text{self}.y - \text{other}.y \\
d = \sqrt{dx^2 + dy^2} \\
\text{return } d
\]
Visualizing a Method Call

\[ P = \text{Point}(3,4) \]
\[ Q = \text{Point}(6,8) \]
\[ D = P.\text{Dist}(Q) \]
Checking Things Out

```python
>>> P1 = RandomPoint(-10,10)
>>> P2 = RandomPoint(-10,10)
>>> M = Midpoint(P1,P2)
>>> print M.Dist(P1)
4.29339610681
>>> print M.Dist(P2)
4.29339610681
```
<table>
<thead>
<tr>
<th>Base Types</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built into Python</td>
<td>Defined in Modules</td>
</tr>
<tr>
<td>Instances are values</td>
<td>Instances are objects</td>
</tr>
<tr>
<td>Instantiate w/ Literals</td>
<td>Instantiate w/ constructors</td>
</tr>
<tr>
<td>Immutable</td>
<td>Mutable</td>
</tr>
</tbody>
</table>
A Note on Copying an Object

There is a difference between creating an alias and creating a genuine second copy of an object.
This Does Not Create a Copy...

```python
>>> Q = Point(3, 4)
>>> P = Q
```

It creates an alias, not a copy.
This Does Create a Copy...

```python
>>> Q = Point(3, 4)
>>> P = Point(Q.x, Q.y)
```
And This Also Creates a Copy...

```python
>>> Q = Point(3,4)
>>> P = copy(Q)
```

The function copy must be imported.
The Module `copy`

```
from copy import copy
```

Import this function and use it to make copies of objects.

`deepcopy` is another useful function from this module—more later.
Using copy

```python
>>> Q = Point(3,4)
>>> P1 = copy(Q)
>>> P1.x = 5
>>> print Q
( 3.000, 4.000)
>>> print P1
( 5.000, 4.000)
```

We are modifying P1, but Q remains the same
Methods vs Functions

It is important to understand the differences between methods and functions, i.e., how they are defined and how they are invoked.
def Dist(P1,P2):
    """ Returns the distance from P1 to P2.
    PreC: P1 and P2 are points
    """
    d = sqrt((P1.x-P2.x)**2+(P1.y-P2.y)**2)
    return d
def Dist(self, other):
    dx = self.x - other.x
    dy = self.y - other.y
    D = sqrt(dx**2 + dy**2)
    return D

>>> P = Point(3, 4)
>>> Q = Point(6, 8)
>>> P.Dist(Q)
5.0

def Dist(P, Q):
    dx = P.x - Q.x
    dy = P.y - Q.y
    D = sqrt(dx**2 + dy**2)
    return D

>>> P = Point(3, 4)
>>> Q = Point(6, 8)
>>> Dist(Q, P)
5.0