Lecture #15: OOP

• Just as `def` defines functions and allows us to extend Python with new operations, `class` defines types and allows us to extend Python with new kinds of data.

• What do we want out of a class?
  - A way of defining named new types of data.
  - A means of defining and accessing state for these objects.
  - A means of defining and using operations specific to these objects.
  - In particular, an operation for initializing the state of an object.
  - A means of creating new objects.

From Last Time

• The Account type illustrated how we do each of these
  
  ```python
  class Account:
      __total_deposits = 0  # Define/initialize a class attribute
      def __init__(self, initial_balance):
          self.__balance = initial_balance  # How to initialize
          Account.__total_deposits += initial_balance
      def deposit(self, amount):
          self.__balance += amount
          Account.__total_deposits += amount
      @staticmethod
      def total_deposits():  # Define a class method.
          return Account.__total_deposits
  
  >>> acct1 = Account(1000)
  >>> acct2 = Account(10000)
  >>> acct1.deposit(300)
  >>> Account.total_deposits()
  11300
  >>> acct1.total_deposits()
  11300
  ```

Class Attributes

• Sometimes, a quantity applies to a type as a whole, not a specific instance.

• For example, with Accounts, you might want to keep track of the total amount deposited from all Accounts.

• This is an example of a class attribute.

Class Attributes in Python

```python
class Account:
    __total_deposits = 0  # Define/initialize a class attribute
    def __init__(self, initial_balance):
        self.__balance = initial_balance  # How to initialize
        Account.__total_deposits += initial_balance
    def deposit(self, amount):
        self.__balance += amount
        Account.__total_deposits += amount
    @staticmethod
    def total_deposits():  # Define a class method.
        return Account.__total_deposits

>>> acct1 = Account(1000)
>>> acct2 = Account(10000)
>>> acct1.deposit(300)
>>> Account.total_deposits()
11300
>>> acct1.total_deposits()
11300
```

Modeling Attributes in Python

• Unlike C++ or Java, Python takes a very dynamic approach.

• Classes and class instances behave rather like environment frames.

```python
def Account:
    __total_deposits = 0
    def __init__(self, initial_balance):
        self.__balance = initial_balance
        Account.__total_deposits += initial_balance
    def deposit(self, amount):
        self.__balance += amount
        Account.__total_deposits += amount

acct1 = Account(1000)
acct2 = Account(10000)
acct1.deposit(300)

>>> acct1 = Account(1000)
>>> acct2 = Account(10000)
>>> acct1.deposit(300)
>>> Account.total_deposits()
11300
>>> acct1.total_deposits()
11300
```

Assigning to Attributes

• Assigning to an attribute of an object (including a class) is like assigning to a local variable: it creates a new binding for that attribute in the object selected from (i.e., referenced by the expression on the left of the dot).

```python
>>> def Value:
...     value = 0
... >>> val1 = Value()
... >>> val2 = Value()
... >>> val1.value = 3
... >>> val1.value
0
>>> Value.value
0
>>> val2.value
3
```

Curved boxes are objects.

Flat-bottomed boxes are class objects.

\( x @ y \): look for \( y \) starting at \( x \)
Methods

 Consider

```python
>>> def Foo:
...     def set(self, x):
...         self.value = x

>>> aFoo = Foo()
>>> aFoo.set(13)  # The first parameter of set is aFoo.
>>> aFoo.value
13
>>> aFoo.set
<bound method Foo.set of ...>
```

• Selection of attributes from objects (other than classes) that were defined as functions in the class does something to those attributes so that they take one fewer parameters: first parameter is bound to the selected-from object.

• Effect of selecting `aFoo.set` is like calling `partial_bind(aFoo, Foo.set)`

```python
def partial_bind(obj, func):
    return lambda x: func(obj, x)
```

Inheritance

• Classes are often conceptually related, sharing operations and behavior.

• One important relation is the subtype or "is-a" relation.

• Examples: A car is a vehicle. A square is a plane geometric figure.

• When multiple types of object are related like this, one can often define operations that will work on all of them, with each type adjusting the operation appropriately.

• In Python (like C++ and Java), a language mechanism called inheritance accomplishes this.

Example: Geometric Plane Figures

• Want to define a collection of types that represent polygons (squares, trapezoids, etc.).

• First, what are the common characteristics that make sense for all polygons?

```python
class Polygon:
    def is_simple(self):
        """True iff I am simple (non-intersecting)."""
    def area(self): ...
    def bbox(self):
        """(xlow, ylow, xhigh, yhigh) of bounding rectangle."""
    def num_sides(self): ...
    def vertices(self):
        """My vertices, ordered clockwise, as a sequence of (x, y) pairs."""
    def describe(self):
        """A string describing me.""
```

• The point here is mostly to document our concept of Polygon, since we don't know how to implement any of these in general.

Partial Implementations

• Even though we don't know anything about Polygons, we can give default implementations.

```python
class Polygon:
    def is_simple(self):
        raise NotImplemented
    def area(self):
        raise NotImplemented
    def vertices(self):
        raise NotImplemented
    def bbox(self):
        V = self.vertices()
        xlow, ylow = xhigh, yhigh = V[0]
        for x, y in V[1:]:
            xlow, ylow = min(x, xlow), min(y, ylow),
            xhigh, yhigh = max(x, xhigh), max(y, yhigh),
        return xlow, ylow, xhigh, yhigh
    def num_sides(self):
        return len(self.vertices())
    def describe(self):
        return "A polygon with vertices {0}".format(self.vertices())
```

Specializing Polygons

• At this point, we can introduce simple (non-intersecting) polygons, for which there is a simple area formula.

```python
class SimplePolygon(Polygon):
    def is_simple(self):
        return True
    def area(self):
        a = 0.0
        V = self.vertices()
        for i in range(len(V)-1):
            a += V[i][0] * V[i+1][1] - V[i+1][0] * V[i][1]
        return -0.5 * a
```

• This says that a SimplePolygon is a kind of Polygon, and that the attributes of Polygon are to be inherited by simple Polygon.

• So far, none of these Polygons are much good, since they have no defined vertices.

• We say that Polygon and SimplePolygon are abstract types.

A Concrete Type

• Finally, a square is a type of simple Polygon:

```python
class Square(SimplePolygon):
    def __init__(self, xll, yll, side):
        """A square with lower-left corner at (xll,yll) and given length on a side.""
        self._x = xll
        self._y = yll
        self._s = side
        def vertices(self):
            x0, y0, s = self._x, self._y, self._s
            return ((x0, y0), (x0, y0+s), (x0+s, y0+s),
                (x0+s, y0), (x0, y0))
        def describe(self):
            return "A {0}x{0} square with lower-left corner {1},{2}".
```

• Don't have to define area, etc., since the defaults work.

• We chose to override describe to give a more specific description.
Inheritance Explained

- Inheritance (in Python) works like nested environment frames.

```
Inheritance Diagram:

SimplePolygon:
- Polygon:
  - is_simple:
  - area:
  - bbox:
  - num_sides:
  - vertices:
  - describe:

Square:
- _init_:
- area:
- vertices:
- describe:

Square(5,6,10)

x: 5
y: 6
s: 10
```

Last modified: Fri Mar 2 00:38:10 2012