

Lecture #15: OOP

Public Service Announcement: Hackers@Berkeley will be hosting a HackJam this Saturday—

- Low-pressure hackathon for both experienced makers and newbies.
- Work together, eat food, and
- Hack something together in just 12 hours.
- Workshops to help you make something cool.
- Judges, prizes, and most importantly - food.
- RSVP by joining the Facebook event page:

<https://www.facebook.com/events/1448019312098352/>

Guerrilla Section #2: Extra groupwork-based section on mastering Recursion. Sunday (March 2nd) at 4pm in 271 Soda (cardkey entry). Check Piazza for details.

Extending the Mutable Objects: Classes

- We've seen a variety of builtin mutable types (sets, dicts, lists).
- ... And a general way of constructing new ones (functions referencing nonlocal variables).
- But in actual practice, we use a different way to construct new types—syntax that leads to clearer programs that are more convenient to read and maintain.
- The Python **class** statement defines new classes or types, creating new, vaguely dictionary-like varieties of object.

Simple Classes: Bank Account

```
class Account: # Type name
    # constructor method
    def __init__(self, initial_balance):
        self._balance = initial_balance

    def balance(self): # instance method
        return self._balance # instance variable

    def deposit(self, amount):
        if amount < 0:
            raise ValueError("negative deposit")
        self._balance += amount

    def withdraw(self, amount):
        if 0 <= amount <= self._balance:
            self._balance -= amount
        else: raise ValueError("bad withdrawal")
```

```
>>> mine = Account(1000)
>>> mine.deposit(100)
>>> mine.balance()
1100
>>> mine.withdraw(200)
>>> mine.balance()
900
```

Class Concepts

- Classes beget *instances*, created by “calling” the class: `Account(1000)`.
- Each such `Account` object (instance) contains *attributes*, accessed using `object.attribute` notation.
- The *defs* inside classes define function-valued attributes called *methods* (full names: `Account.balance`, etc.) Each object has a copy.
- A call `mine.deposit(100)` is essentially `Account.deposit(mine, 100)`.
- By convention, we therefore call the first argument of a method something like “self” to indicate that it is the object from which we got the method.
- When an object is created, the special `__init__` method is called first.
- Each `Account` object has other attributes (`_balance`), which we create by assignment, again using dot notation.

Philosophy

- 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.

Applied Philosophy

- The Account type illustrates how we do each of these

<code>class Account:</code>	Define named new type
<code> def __init__(self, initial_balance):</code>	How to initialize
<code> self._balance = initial_balance</code>	Create/modify state
<code> def balance(self):</code>	Define new operation on Accounts
<code> return self._balance</code>	Access state of an Account
<code> ...</code>	
<code>myAccount = Account(1000)</code>	Create a new Account object,
<code>print(myAccount.balance())</code>	Operate on an Account object.

Class Attributes

- Things like `_balance`, `__init__`, and `deposit` are attributes of *instances of classes*.
- Sometimes, a quantity applies to a class 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

```
class Account:
    _total_deposits = 0      # Define/initialize a class attribute
    def __init__(self, initial_balance):
        self._balance = initial_balance
        Account._total_deposits += initial_balance # Use the class name
    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.

```
def Account:  
    _total_deposits = 0
```

```
def __init__(...):  
    self._balance = ...  
    Account._total_deposits = ...
```

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

Account:

_total_deposits: 11300
__init__: ...
balance: ...
deposit: ...
withdraw: ...

acct1:

_balance: 1300

acct2:

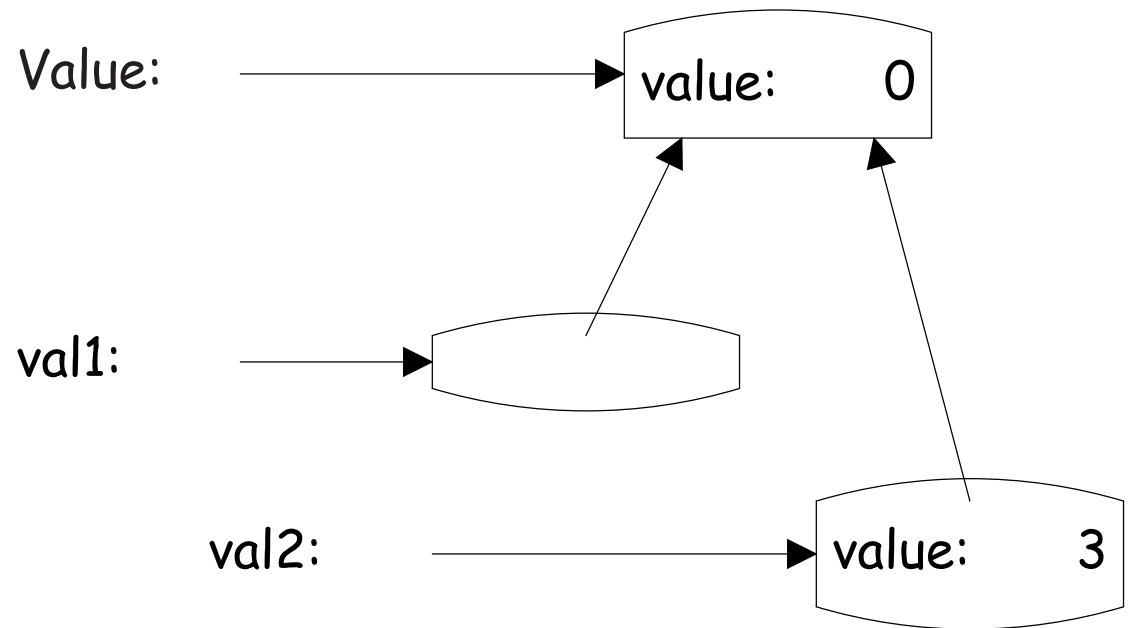
_balance: 10000

- Curved boxes are objects.
- Flat-bottomed boxes are class objects.
- 'x.y': look for 'y' starting at 'x'

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).

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



Methods

- Consider

```
>>> 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)`, where

```
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?

```
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.

```
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.

```
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:

```
class Square(SimplePolygon):
    def __init__(self, x11, y11, side):
        """A square with lower-left corner at (x11,y11) and
        given length on a side."""
        self._x = x11
        self._y = y11
        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})" \
            .format(self._s, self._x, self._y)
```

- 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.

