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

```
class Account:                          Define named new type

    def __init__(self, initial_balance):  How to initialize
        self._balance = initial_balance  Create/modify state

    def balance(self):                   Define new operation on Accounts
        return self._balance   Access state of an Account

    ...

myAccount = Account(1000)          Create a new Account object,
print(myAccount.balance())         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
```

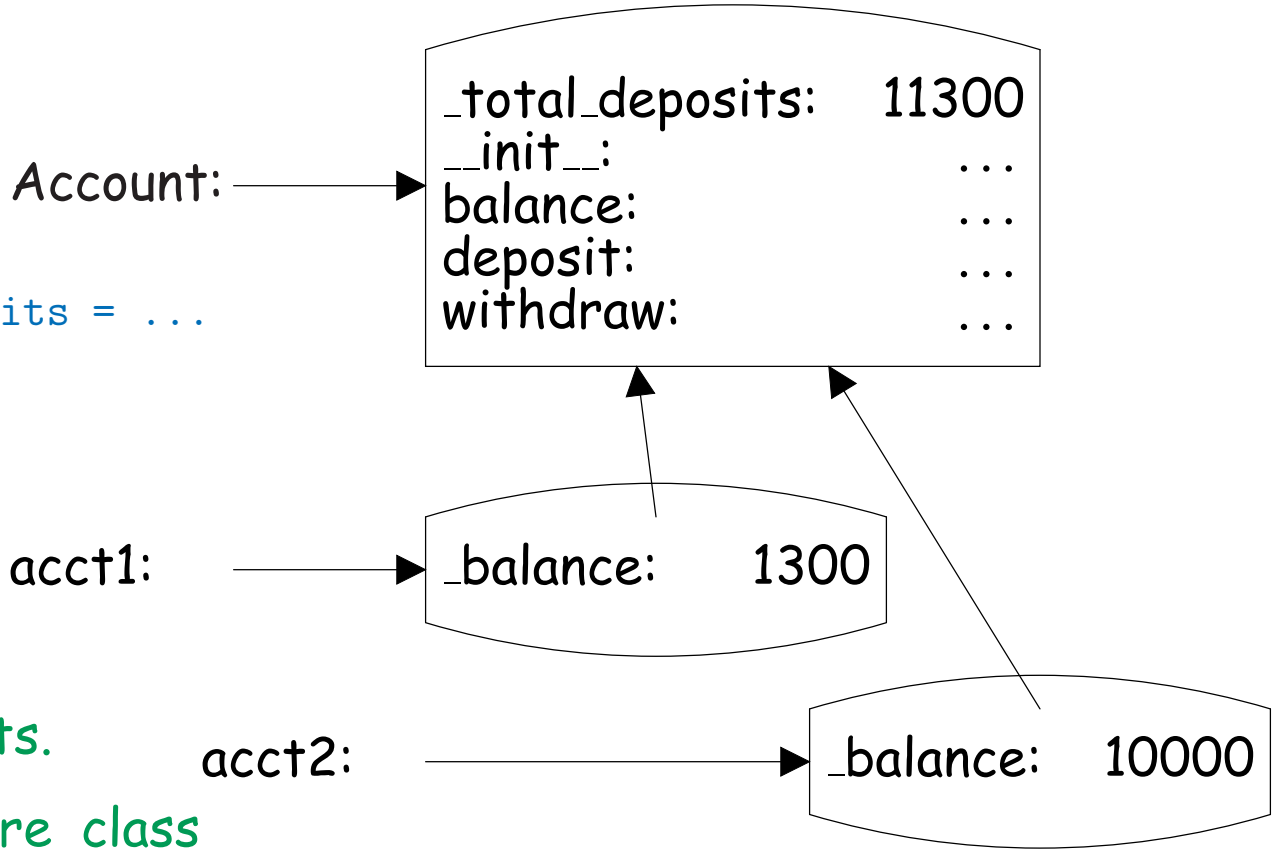
Modeling Attributes in Python

- 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)
```

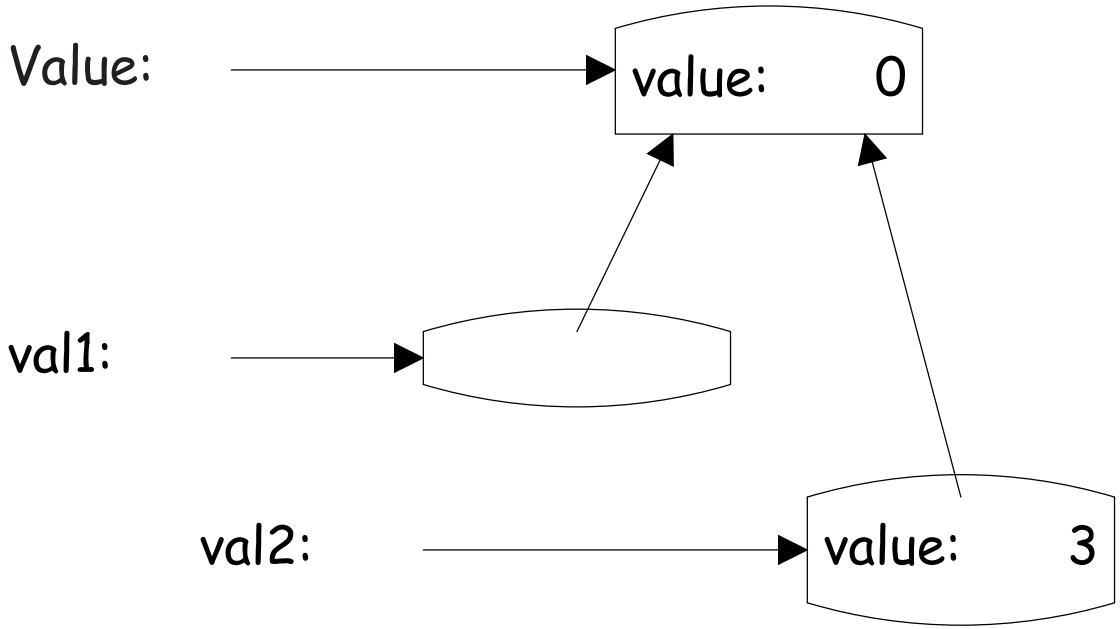


- Curved boxes are objects.
- Flat-bottomed boxes are class objects.
- 'x.y': look for 'y' starting at 'x'

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)
```