Lecture 15: Object-Oriented Programming

Brian Hou
July 18, 2016

Announcements

- Homework 6 is due 7/20 at 11:59pm
- Project 3 is due 7/26 at 11:59pm
  - Earn 1 EC point for completing it by 7/25
- Quiz 5 on 7/21 at the beginning of lecture
  - May cover mutability, object-oriented programming
- Midterm grades are released, regrade requests due tonight

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

- This week (Objects), the goals are:
  - To learn the paradigm of *object-oriented programming*
  - To study applications of, and problems that be solved using, OOP

Previously, on CS 61A...

- We defined our own data types!
  - Rational numbers, dictionaries, linked lists, trees
- Data abstraction helped us manage the complexity of using these new data types
  - Separated their *usage* from their underlying *implementation*
- We defined operations for these data types:
  - len_link, getitem_link, contains_link, map_link...
- Problems?
  - Abstraction violations
  - Program organization

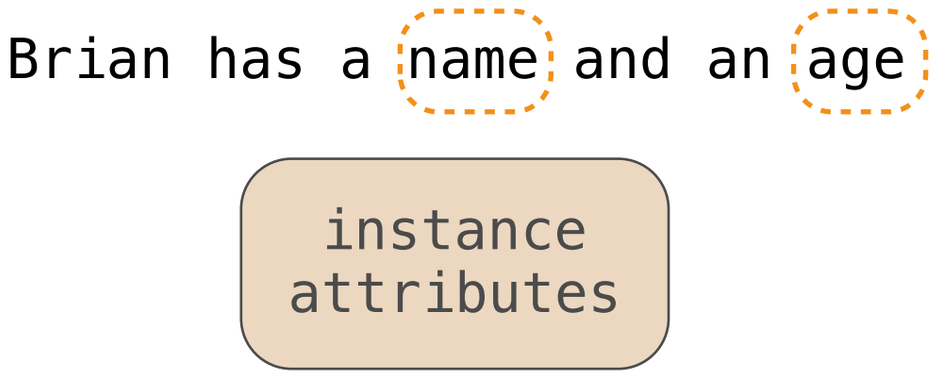
Object-Oriented Programming

Object-Oriented Programming

- A new programming paradigm: think in terms of *objects*
  - Objects have attributes and can take actions
  - Objects can interact with each other
- Computations are the result of interactions between objects

Classes

- Every object is an *instance* of a *class*
- A class is a type or a category of objects (often capitalized)
- A class provides a blueprint for its objects



The Account Class

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance

```
>>> a = Account('Brian')
>>> a.balance
0
>>> a.holder
'Brian'
```

Idea: All bank accounts should have withdraw and deposit behaviors that all work in the same way

Better idea: All bank accounts share a withdraw method and a deposit method

```
>>> a.deposit(15)
15
>>> a.balance
15
>>> a.withdraw(10)
5
>>> a.balance
5
>>> a.withdraw(10)
'Insufficient funds'
```

The Class Statement

```
class <name>:
    <suite>
```

- When executing a **class** statement, Python creates a new frame and executes the statements in <suite> (typically assignment and **def** statements)
- Once all the statements in <suite> have been executed, a new class with those bindings is created and bound to <name> in the first frame of the original environment

Constructing Objects

Idea: All bank accounts have a balance and an account holder

```
>>> a = Account('Brian')
>>> a.balance
0
>>> a.holder
'Brian'
```

```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```

`__init__` is called a constructor

An account instance

balance: 0 holder: 'Brian'

When a class is called:

- A new instance of that class is created
- The special **`__init__`** method of the class is called with the new instance as its first argument (named **self**), along with any additional arguments provided in the call expression

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('Brian')
>>> b = Account('Marvin')
>>> a.holder
'Brian'
>>> b.holder
'Marvin'
>>> a is b
False
```

Every call to Account creates a new Account instance.

Binding an object to a new name using assignment does not create a new object:

```
>>> c = a
>>> c is a
True
```

Methods

Methods

- Methods are functions defined within a **class** statement
- These **def** statements create function objects as always, but their names are bound as attributes of the class

```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

self should always be bound to an instance of the Account class

Invoking Methods (demo)

- All methods have access to the object via the self parameter, and so they can all access and manipulate the object's state

```
class Account:
    ...
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Dot notation automatically passes the first argument to a method

```
>>> a1 = Account('Brian')
>>> a1.deposit(100)
100
```

Bound to self

Invoked with one argument

```
>>> a2 = Account('Brian')
>>> Account.deposit(a2, 100)
100
```

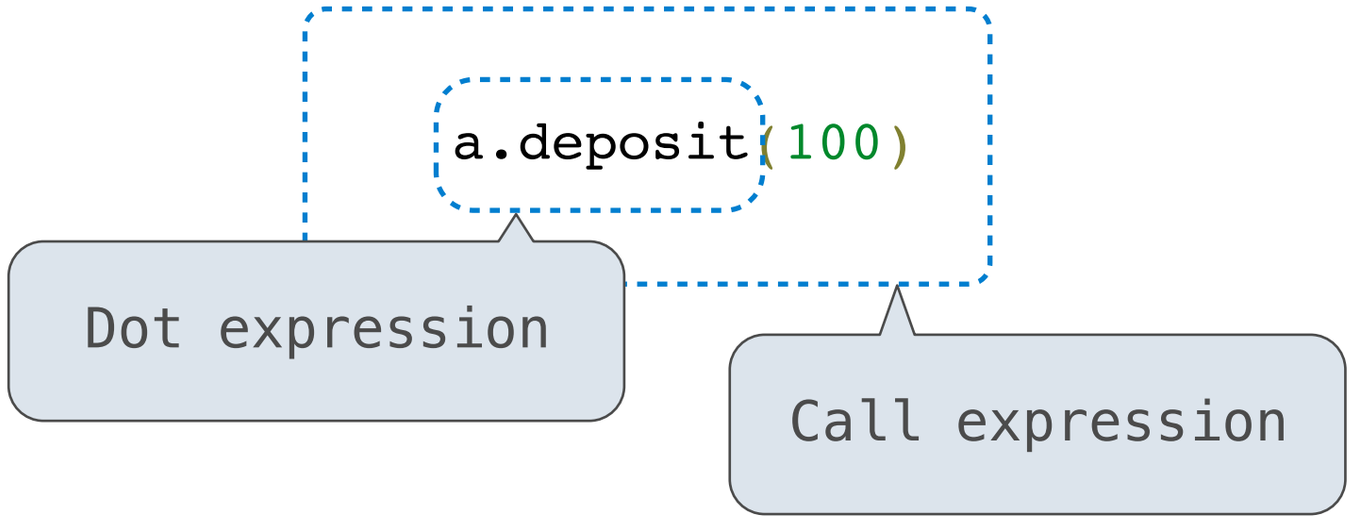
Invoked with two arguments

Attributes

Dot Notation

<expr>.<name>

- Dot notation accesses attributes of an instance or its class
- <expr> can be any valid Python expression
- Look up the value of <name> in the object <expr>



Accessing Attributes (demo)

<expr>.<name>

- The built-in getattr function does the same thing as dot expressions
  - a.balance is equivalent to getattr(a, 'balance')
  - a.deposit is equivalent to getattr(a, 'deposit')
    - a.deposit(100) is equivalent to getattr(a, 'deposit')(100)
- The built-in hasattr function returns whether an object has an attribute with that name
- Accessing an attribute in an object may return:
  - One of its instance attributes, or
  - One of the attributes of its class

Methods and Functions (demo)

- Python distinguishes between:
  - *Functions*, which we have been creating since the beginning of the course
  - *Bound methods*, which combines a function and the instance on which that function will be invoked

```
>>> a = Account('Brian')
>>> type(Account.deposit)
<class 'function'>
>>> type(a.deposit)
<class 'method'>
>>> Account.deposit(a, 100)
100
>>> a.deposit(100)
200
```

Function: all arguments are within parentheses

Method: one argument (self) before the dot and other arguments within parentheses

Class Attributes (demo)

- Class attributes are "shared" across all instances of a class because they are attributes of the class, not the instance

```
class Account:
    interest = 0.02
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```

```
>>> a = Account('Brian')
>>> b = Account('Marvin')
>>> a.interest
0.02
>>> b.interest
0.02
```

The **interest** attribute is *not* part of the instance; it's part of the class!

Evaluating Dot Expressions

<expr>.<name>

- Evaluate <expr>, which yields the object of the dot expression
- <name> is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned
- If not, <name> is looked up in the class, which yields a class attribute value
- That value is returned unless it is a function, in which case a bound method is returned instead

Break!

Inheritance

Inheritance

- Inheritance is a technique for relating classes together
- Common use: a *specialized* class inherits from a more *general* class

```
class <new class>(<base class>):
    ...
```

- The new class *shares* attributes with the base class (inherits attributes of its base class)
- The new class *overrides* certain inherited attributes
- Implementing the new class is now as simple as specifying how it’s *different* from the base class

Inheritance Example

```
class Account:
    """A bank account."""
    ...
```

- Bank accounts have:
  - an account holder
  - a balance
  - an interest rate of 2%
- You can:
  - deposit to an account
  - withdraw from an account

```
class CheckingAccount(Account):
    """A checking account."""
    ...
```

- Checking accounts have:
  - an account holder
  - a balance
  - an interest rate of 1%
  - a withdrawal fee of $1
- You can:
  - deposit to an account
  - withdraw from an account (but there's a fee!)

Inheritance Example (demo)

```
class Account:
    """A bank account."""
    ...
```

- Bank accounts have:
  - an account holder
  - a balance
  - an interest rate of 2%
- You can:
  - deposit to an account
  - withdraw from an account

```
class CheckingAccount(Account):
    """A checking account."""
    ...
```

- Checking accounts have:
  - an account holder
  - a balance
  - an interest rate of **1%**
  - **a withdrawal fee of $1**
- You can:
  - deposit to an account
  - withdraw from an account **(but there's a fee!)**

Attribute Lookup on Classes (demo)

Base class attributes *aren't* copied into subclasses!

To look up a name in a class:

1. If it is an attribute in the class, return that value.
2. Otherwise, look up the name in the base class, if one exists

```
>>> ch = CheckingAccount('Marvin')    # Account.__init__
>>> ch.interest                       # Found in CheckingAccount
0.01
>>> ch.deposit(20)                    # Found in Account
20
>>> ch.withdraw(5)                    # Found in CheckingAccount
14
```

Designing for Inheritance

- Don't repeat yourself; use existing implementations
- Attributes that have been overridden are still accessible via class objects
- Look up attributes on instances whenever possible

```
class CheckingAccount(Account):
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self,
                                amount + self.withdraw_fee)
```

Attribute look-up on base class

Preferred to CheckingAccount.withdraw_fee to allow for further specialization

Summary

- Object-oriented programming is another way (paradigm) to organize and reason about programs
- Computations are the result of interactions between objects
- The Python class statement allows us to create user-defined data types that can be used just like built-in data types
- Inheritance is a powerful tool for further extending these user-defined data types