Inheritance

Announcements

Attributes

Methods and Functions

Python distinguishes between:
- Functions, which we have been creating since the beginning of the course, and
- Bound methods, which couple together a function and the object on which that method will be invoked

Object + Function = Bound Method

```python
>>> type(Account.deposit)
<class 'function'>
```

```python
>>> type(tom_account.deposit)
<class 'method'>
```

```python
>>> Account.deposit(tom_account, 1001)
1011
```

```python
>>> tom_account.deposit(1004)
2015
```

Function: all arguments within parentheses
Method: One object before the dot and other arguments within parentheses

Class Attributes

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

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

# Additional methods would be defined here
```

```python
>>> tom_account = Account('Tom')
```

```python
>>> jim_account = Account('Jim')
```

```python
>>> tom_account.interest
0.02
```

```python
>>> jim_account.interest
0.02
```

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs
Classes are objects too, so they have attributes
Instance attribute: attribute of an instance
Class attribute: attribute of the class of an instance

```
Terminology:

Python object system:

Functions are objects
Bound methods are also objects: a function that has its first parameter "self" already bound to an instance
Dot expressions evaluate to bound methods for class attributes that are functions
```

```
Looking Up Attributes by Name

Expression . name
```

To evaluate a dot expression:
1. Evaluate the expression to the left of the dot, which yields the object of the dot expression
2. name is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned
3. If not, name is looked up in the class, which yields a class attribute value
4. That value is returned unless it is a function, in which case a bound method is returned instead

```
Class Attributes

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

```

```
Attribute Assignment

The interest attribute is not part of the instance; it's part of the class!
```
Assignment to Attributes

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression:

* If the object is an instance, then assignment sets an instance attribute
* If the object is a class, then assignment sets a class attribute

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

Instance tom_account:
  tom_account.interest = 0.08
```

The object of that dot expression is the instance or class that is assigned to.

```
Assignment statements with a dot expression on their left-hand side affect attributes for
the object of that dot expression:

If the object is a class, then assignment sets a class attribute
If the object is an instance, then assignment sets an instance attribute

```
Instance attributes of tom_account:
  tom_account.interest = 0.08
```

```
Inheritance Example

A CheckingAccount is a specialized type of Account

```
CheckingAccount(Account):
  # A bank account that charges for withdrawals.
  # Interest is 0.05

  def withdraw(self, amount):
    return Account.withdraw(self, amount) + self.withdraw_fee

```

Inheritance

Inheritance is a technique for relating classes together.

A common use: Two similar classes differ in their degree of specialization.

The specialized class may have the same attributes as the general class, along with some special-case behavior.

```
class CheckingAccount(Account):
  # A bank account that charges for withdrawals.
  # Interest is 0.05

  def withdraw(self, amount):
    return Account.withdraw(self, amount) + self.withdraw_fee
```

Looking Up Attribute Names on Classes

Base class attributes aren’t copied into subclasses!

To look up a name in a class:
1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

```
Inheritance Example

A CheckingAccount is a specialized type of Account

>>> ch = CheckingAccount('Tom')
>>> ch.interest # Lower interest rate for checking accounts
0.02
>>> ch.deposit(20) # Deposits are the same
20
>>> ch.withdraw(5) # Withdrawals incur a $1 fee
14
```

```
Inheritance

Inheritance is a technique for relating classes together.

A common use: Two similar classes differ in their degree of specialization.

The specialized class may have the same attributes as the general class, along with some special-case behavior.

```
class CheckingAccount(Account):
  # A bank account that charges for withdrawals.
  # Interest is 0.05

  def withdraw(self, amount):
    return Account.withdraw(self, amount) + self.withdraw_fee
```

```
Looking Up Attribute Names on Classes

Base class attributes aren’t copied into subclasses!

To look up a name in a class:
1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

```
Inheritance Example

A CheckingAccount is a specialized type of Account

>>> ch = CheckingAccount('Tom') # Calls Account.__init__
>>> ch.interest # Found in CheckingAccount
0.02
>>> 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.

```
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):
  # A bank account that charges for withdrawals.
  # Interest is 0.05

  def withdraw(self, amount):
    return Account.withdraw(self, amount) + self.withdraw_fee
```
Inheritance and Composition

Object-oriented programming shines when we adopt the metaphor of objects and classes. Inheritance is best for representing is-a relationships:

- E.g., a checking account is a specific type of account
- So, CheckingAccount inherits from Account

Composition is best for representing has-a relationships:

- E.g., a bank has a collection of bank accounts it manages
- So, A bank has a list of accounts as an attribute

Multiple Inheritance

A class may inherit from multiple base classes in Python:

class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
    def __init__(self, account_holder):
        self.holder = account_holder
        self.balance = 1  # A free dollar!

>>> such_a_deal = AsSeenOnTVAccount('John')
>>> such_a_deal.balance
1

Resolving Ambiguous Class Attribute Names

Account

CheckingAccount

SavingsAccount

AsSeenOnTVAccount

Instance attribute

>>> such_a_deal = AsSeenOnTVAccount('John')
>>> such_a_deal.balance
1

SavingsAccount method

>>> such_a_deal.deposit(20)
10

CheckingAccount method

>>> such_a_deal.withdraw(5)
13

Complicated Inheritance

Attributes Lookup Practice

Inheritance and Attribute Lookup

Inheritance and Composition
Biological Inheritance

Moral of the story: Inheritance can be complicated, so don't overuse it!