Announcements

Attributes

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.

Methods

Terminology: Python object system:

* `<instance>.<method_name>`

Reminder: Looking Up Attributes by 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.

Attribute Assignment

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.

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.

<table>
<thead>
<tr>
<th>Class Attributes</th>
<th>Instance Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>account.interest = 0.88</td>
<td>tom_account.interest = 0.08</td>
</tr>
<tr>
<td>holder = holder</td>
<td>holder</td>
</tr>
<tr>
<td>balance = 0</td>
<td>balance = 0</td>
</tr>
<tr>
<td></td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>0.02</td>
</tr>
</tbody>
</table>

Attribute Assignment Statements

<table>
<thead>
<tr>
<th>Account class attributes</th>
<th>Instance attributes of jim_account</th>
<th>Instance attributes of tom_account</th>
</tr>
</thead>
<tbody>
<tr>
<td>interest: 0.05</td>
<td>balance: 0</td>
<td>holder: 'Jim'</td>
</tr>
<tr>
<td>withdraw, deposit, <strong>init</strong></td>
<td>balance = 0</td>
<td>holder: 'Tom'</td>
</tr>
<tr>
<td>jim_account = Account('Jim')</td>
<td>&gt;&gt;&gt; jim_account = Account('Jim')</td>
<td>&gt;&gt;&gt; jim_account = Account('Jim')</td>
</tr>
<tr>
<td>&gt;&gt;&gt; jim_account.balance = 0.05</td>
<td>&gt;&gt;&gt; jim_account.balance = 0.05</td>
<td>&gt;&gt;&gt; jim_account.balance = 0.05</td>
</tr>
<tr>
<td>&gt;&gt;&gt; jim_account.deposit(100)</td>
<td>&gt;&gt;&gt; jim_account.deposit(100)</td>
<td>&gt;&gt;&gt; jim_account.deposit(100)</td>
</tr>
<tr>
<td>&gt;&gt;&gt; jim_account.withdraw(50)</td>
<td>&gt;&gt;&gt; jim_account.withdraw(50)</td>
<td>&gt;&gt;&gt; jim_account.withdraw(50)</td>
</tr>
<tr>
<td>&gt;&gt;&gt; jim_account.interest = 0.06</td>
<td>&gt;&gt;&gt; jim_account.interest = 0.06</td>
<td>&gt;&gt;&gt; jim_account.interest = 0.06</td>
</tr>
<tr>
<td>&gt;&gt;&gt; jim_account.holder</td>
<td>&gt;&gt;&gt; jim_account.holder</td>
<td>&gt;&gt;&gt; jim_account.holder</td>
</tr>
</tbody>
</table>
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 <Name>(<Base Class>):
    <suite>
```

Conceptually, the new subclass inherits attributes of its base class.

The subclass may override certain inherited attributes.

Using inheritance, we implement a subclass by specifying its differences from the the base class.

```
return super().withdraw(amount + self.withdraw_fee)
```

Inheritance Example

A `CheckingAccount` is a specialized type of `Account`.

```
ch = CheckingAccount('Tom')
>>> ch.interest
# Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```

Most behavior is shared with the base class `Account`.

```
class CheckingAccount(Account):
    '''A bank account that charges for withdrawals.'''
    withdraw_fee = 1
    interest = 0.01

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.

```
ch = CheckingAccount('Tom')  # Calls Account.__init__
>>> ch.interest
# Found in CheckingAccount
0.01
>>> ch.deposit(20)  # Found in Account
20
```

Attribute look-up on base class

Preferred to `CheckingAccount.withdraw_fee` to allow for specialized accounts.

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.'''
    withdraw_fee = 1
    interest = 0.01

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

Object-Oriented Design

Object-oriented programming shines when we adopt the metaphor.

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

Inheritance and Composition

Object-oriented programming shines when we adopt the metaphor.

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

(Demo)
Multiple Inheritance

```python
class SavingsAccount(Account):
    deposit_fee = 2
    def deposit(self, amount):
        return Account.deposit(self, amount - self.deposit_fee)

A class may inherit from multiple base classes in Python.

CleverBank marketing executive has an idea:
• Low interest rate of 1%
• A $1 fee for withdrawals
• A $2 fee for deposits
• A free dollar when you open your account

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
>>> such_a_deal.deposit(20)
19
>>> such_a_deal.withdraw(5)
13
```

Resolving Ambiguous Class Attribute Names

```mermaid
graph LR
    Account --> CheckingAccount
    Account --> SavingsAccount
    AsSeenOnTVAccount --> Instance attribute
    AsSeenOnTVAccount --> SavingsAccount method
    AsSeenOnTVAccount --> CheckingAccount method
```

Complicated Inheritance

```mermaid
graph LR
    Grandma[Grandma] --> Grandpa[Grandpa]
    Grandma[Grandma] --> Grandaddy[Grandaddy]
    Dad[You] --> Half Cousin[some_other_guy]
    Half Cousin[some_other_guy] --> Double Half Uncle[double_guy]
    Half Cousin[some_other_guy] --> Double Half Uncle[double_guy]
    Dad[You] --> Aunt[some_guy]
    Aunt[some_guy] --> Half Cousin[some_guy]
    Aunt[some_guy] --> Double Half Uncle[double_guy]
    Half Cousin[some_guy] --> Double Half Uncle[double_guy]

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