Lecture 16

Wednesday, October 3

Attributes

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

Methods

- 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.

Terminology: Attributes, Functions, and Methods

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

Looking Up Attributes by Name (Abbreviated)

- To evaluate a dot expression:
  1. Evaluate the `<expression>`.
  2. `<name>` is matched against the instance attributes.
  3. If not found, `<name>` is looked up in the class.
  4. That class attribute value is returned unless it is a function, in which case a bound method is returned.

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.

```
Instance Attribute Assignment

Instance Account
| balance: 0 | holder: 'Jim' |
| interest: 0.08 |

tom_account = Account('Tom')
>>> tom_account
Account('Tom', 0.02)

But the name ('interest') is not looked up

Class Attribute Assignment

Class Account
| balance: 0 | holder: 'Tom' |
| interest: 0.04 |

Account = Account('Jim')
>>> Account
Account('Jim', 0.08)

Attribute Assignment Statements

- `<expression> . <name>`

    `tom_account = Account('Tom')`
    >>> tom_account
    Account('Tom', 0.02)

    `jim_account = Account('Jim')`
    >>> jim_account
    Account('Jim', 0.08)

    `tom_account.interest = 0.08`
    >>> tom_account
    Account('Tom', 0.08)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `tom_account.interest = 0.08`
    >>> tom_account
    Account('Tom', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)

    `tom_account.interest = 0.05`
    >>> tom_account
    Account('Tom', 0.05)

    `jim_account.interest = 0.08`
    >>> jim_account
    Account('Jim', 0.08)

    `Account.interest = 0.05`
    >>> Account
    Account(0.05)
Inheritance

A technique for relating classes together

Common use: Similar classes differ in amount of specialization

Two classes have overlapping attribute sets, but one represents a special case of the other.

Conceptually, the new subclass "shares" attributes with its base class.

The subclass may override certain inherited attributes.

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

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.

Designing for Inheritance: General Base Classes

Base classes may contain logic that is meant for subclasses.

Example: Same CheckingAccount behavior; different approach

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

```python
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)
```

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.

```python
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)
```

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 Account instances as an attribute.

No local state at all? Just write a pure function!
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 wants:

• Low interest rate of 1%
• A $1 fee for withdrawals
• A $2 fee for deposits
• A free dollar when you open your account

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

```python
>>> 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
diagram concept map
    Account
    CheckingAccount
    SavingsAccount
    AsSeenOnTVAccount
```

```mermaid
mermaid
    diagram graph LR
    "Account"[label=instance attr]
    "CheckingAccount"[label=method]
    "SavingsAccount"[label=method]
```

Human Relationships

```mermaid
mermaid
    diagram concept map
    Some_Guy - Grandpa
    Grandpa - Grammy
    Som_Dude - Double
```