CS61A Lecture 20
Amir Kamil and Julia Oh
UC Berkeley
March 8, 2013

Announcements
- HW7 due on Wednesday
- Ants project out

Dot Expressions
Objects receive messages via dot notation
Dot notation accesses attributes of the instance or its class
\[
\text{<expression> . <name>}
\]
The \text{<expression>} can be any valid Python expression
The \text{<name>} must be a simple name
Evaluates to the value of the attribute \text{looked up} by \text{<name>}
in the object that is the value of the \text{<expression>}

Accessing Attributes
Using \text{getattr}, we can look up an attribute using a string, just
as we did with a dispatch function/dictionary

\[
\begin{align*}
\text{>>> } & \text{getattr(tom_account, 'balance')} \\
& 10 \\
\text{>>> } & \text{hasattr(tom_account, 'deposit')} \\
& \text{True}
\end{align*}
\]
\text{getattr} and dot expressions look up a name in the same way
Looking up an attribute name in an object may return:
- One of its instance attributes, or
- One of the attributes of its class

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

\[
\text{Object + Function = Bound Method}
\]

\[
\begin{align*}
\text{>>> } & \text{type(Account.deposit)} \\
& \text{<class 'function'>} \\
\text{>>> } & \text{type(tom_account.deposit)} \\
& \text{<class 'method'>} \\
\text{>>> } & \text{Account.deposit(tom_account, 1001)} \\
& 1011 \\
\text{>>> } & \text{tom_account.deposit(1000)} \\
& 2011
\end{align*}
\]

Methods and Currying
Earlier, we saw \text{currying}, which converts a function that takes in
multiple arguments into multiple chained functions.
The same procedure can be used to create a bound method
from a function

\[
\begin{align*}
\text{def curry(f):} \\
& \text{def outer(x):} \\
& \text{def inner(*args):} \\
& \text{return f(x, *args)} \\
\text{return outer}
\end{align*}
\]

\[
\begin{align*}
\text{>>> } & \text{add2 = curry(add)(2)} \\
& 5 \\
\text{>>> } & \text{tom_deposit = curry(Account.deposit)(tom_account)} \\
\text{>>> } & \text{tom_deposit(1000)} \\
& 3011
\end{align*}
\]
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

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 on instances evaluate to bound methods for class attributes that are functions.

Looking Up Attributes by Name

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.

Class Attributes

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

class Account(object):
    interest = 0.02  # Class attribute
    def __init__(self, account_holder):
        self.balance = 0  # Instance attribute
        self.holder = account_holder
        # Additional methods would be defined here
    >>> tom_account = Account('Tom')
    >>> jim_account = Account('Jim')
    >>> tom_account.interest = 0.02
    >>> jim_account.interest = 0.02

    interest is not part of the instance that was somehow copied from 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

Instance Variable Assignment:
Account.interest = 0.04

Attribute Assignment Statements

Account class attributes
interest: 0.05
(withdraw, deposit, __init__)

Balance: 0
holder: 'Jim'
interest: 0.08

>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest = 0.02
>>> jim_account.interest = 0.02
>>> tom_account.interest = 0.04
>>> account.interest = 0.04
>>> tom_account.interest = 0.04

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.

class <name>(<base class>):
    <suite>

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.
Inheritance Example

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

```python
>>> 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)
```

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.

```python
>>> ch = CheckingAccount('Tom') # Calls 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.

```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!
```

General Base Classes

Base classes may contain logic that is meant for subclasses.

Example: Same `CheckingAccount` behavior; different approach

```python
class Account(object):
    withdraw_fee = 0
    def withdraw(self, amount):
        amount += self.withdraw_fee
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

```python
class CheckingAccount(Account):
    interest = 0.01
    withdraw_fee = 1
```

Preferable alternative to `CheckingAccount.withdraw_fee`

```python
Attribute look-up on base class
```
Multiple Inheritance

A class may inherit from multiple base classes in Python.

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

Resolving Ambiguous Class Attribute Names

```python
Instance attribute
SavingsAccount method
CheckingAccount method
Account
CheckingAccount
SavingsAccount
AsSeenOnTVAccount

Instance attribute
SavingsAccount method
CheckingAccount method

>>> such_a_deal = AsSeenOnTVAccount("John")
>>> such_a_deal.balance
1
>>> such_a_deal.deposit(20)
19
>>> such_a_deal.withdraw(5)
13
```

Human Relationships

- Some_Guy
- Grandma
- Grandpa
- Grandaddy
- Gramammy
- Aunt
- Double
- Half
- Mom
- Dad
- Some_Dude
- Double
- Half
- Cousin
- Double
- Half
- Uncle
- You