61A Extra Lecture 6
Announcements
Announcements

Homework 1 due Monday 10/12 (today)
Announcements

Homework 1 due Monday 10/12 (today)

Homework 2 released next Monday 10/19 is due 11/2
Announcements

Homework 1 due Monday 10/12 (today)

Homework 2 released next Monday 10/19 is due 11/2

Homework 3 is to complete an extension to Project 4
Announcements

Homework 1 due Monday 10/12 (today)

Homework 2 released next Monday 10/19 is due 11/2

Homework 3 is to complete an extension to Project 4
- Due at the same time as Project 4!
Announcements

Homework 1 due Monday 10/12 (today)

Homework 2 released next Monday 10/19 is due 11/2

Homework 3 is to complete an extension to Project 4
• Due at the same time as Project 4!

Homework 4 Released Monday 11/9, Due 11/23
Implementing an Object System
Implementing an Object System

Today's topics:
Implementing an Object System

Today's topics:
• What is a class?
Implementing an Object System

Today's topics:
• What is a class?
• What is an instance?
Implementing an Object System

Today's topics:
• What is a class?
• What is an instance?
• How do we create inheritance relationships?
Implementing an Object System

Today's topics:
• What is a class?
• What is an instance?
• How do we create inheritance relationships?
• How do we write code for attribute look-up procedures?
Implementing an Object System

Today's topics:
- What is a class?
- What is an instance?
- How do we create inheritance relationships?
- How do we write code for attribute look-up procedures?

Tools we'll use:
Implementing an Object System

Today's topics:
- What is a class?
- What is an instance?
- How do we create inheritance relationships?
- How do we write code for attribute look-up procedures?

Tools we'll use:
- Dispatch dictionaries
Implementing an Object System

Today's topics:
• What is a class?
• What is an instance?
• How do we create inheritance relationships?
• How do we write code for attribute look-up procedures?

Tools we'll use:
• Dispatch dictionaries
• Higher-order functions
The OOP Abstraction Barrier (a.k.a. the Line)
The OOP Abstraction Barrier (a.k.a. the Line)
The OOP Abstraction Barrier (a.k.a. the Line)

Above the Line:
The OOP Abstraction Barrier (a.k.a. the Line)

Above the Line:

- Objects with **local state** & interact via **message passing**
The OOP Abstraction Barrier (a.k.a. the Line)

Above the Line:
- Objects with local state & interact via message passing
- Objects are instantiated by classes, which are also objects
The OOP Abstraction Barrier (a.k.a. the Line)

Above the Line:
- Objects with **local state** & interact via **message passing**
- Objects are **instantiated** by classes, which are also objects
- Classes may **inherit** from other classes to share behavior
The OOP Abstraction Barrier (a.k.a. the Line)

Above the Line:
- Objects with local state & interact via message passing
- Objects are instantiated by classes, which are also objects
- Classes may inherit from other classes to share behavior
- Mechanics of objects are governed by "evaluation procedures"
The OOP Abstraction Barrier (a.k.a. the Line)

Above the Line:
- Objects with local state & interact via message passing
- Objects are instantiated by classes, which are also objects
- Classes may inherit from other classes to share behavior
- Mechanics of objects are governed by "evaluation procedures"

Below the Line:
The OOP Abstraction Barrier (a.k.a. the Line)

Above the Line:
- Objects with **local state** & interact via **message passing**
- Objects are **instantiated** by classes, which are also objects
- Classes may **inherit** from other classes to share behavior
- Mechanics of objects are governed by "**evaluation procedures**"

Below the Line:
- Objects have **mutable dictionaries** of attributes
The OOP Abstraction Barrier (a.k.a. the Line)

Above the Line:
- Objects with **local state** & interact via **message passing**
- Objects are **instantiated** by classes, which are also objects
- Classes may **inherit** from other classes to share behavior
- Mechanics of objects are governed by "**evaluation procedures**"

Below the Line:
- Objects have **mutable dictionaries** of attributes
- **Attribute look-up for instances** is a function
The OOP Abstraction Barrier (a.k.a. the Line)

Above the Line:
• Objects with **local state** & interact via **message passing**
• Objects are **instantiated** by classes, which are also objects
• Classes may **inherit** from other classes to share behavior
• Mechanics of objects are governed by "**evaluation procedures**"

Below the Line:
• Objects have **mutable dictionaries** of attributes
• **Attribute look-up for instances** is a function
• **Attribute look-up for classes** is another function
The OOP Abstraction Barrier (a.k.a. the Line)

Above the Line:
• Objects with **local state** & interact via **message passing**
• Objects are **instantiated** by classes, which are also objects
• Classes may **inherit** from other classes to share behavior
• Mechanics of objects are governed by "**evaluation procedures**"

Below the Line:
• Objects have **mutable dictionaries** of attributes
• **Attribute look-up for instances** is a function
• **Attribute look-up for classes** is another function
• **Object instantiation** is another function
Implementing the Object Abstraction
Implementing the Object Abstraction

Fundamental OOP concepts:
Implementing the Object Abstraction

**Fundamental OOP concepts:**
- Object instantiation and initialization
Implementing the Object Abstraction

Fundamental OOP concepts:
• Object instantiation and initialization
• Attribute look-up and assignment
Implementing the Object Abstraction

**Fundamental OOP concepts:**

- Object instantiation and initialization
- Attribute look-up and assignment
- Method invocation
Implementing the Object Abstraction

**Fundamental OOP concepts:**
- Object instantiation and initialization
- Attribute look-up and assignment
- Method invocation
- Inheritance
Implementing the Object Abstraction

**Fundamental OOP concepts:**
- Object instantiation and initialization
- Attribute look-up and assignment
- Method invocation
- Inheritance

**Not-so-fundamental issues (that we'll skip):**
Implementing the Object Abstraction

**Fundamental OOP concepts:**
- Object instantiation and initialization
- Attribute look-up and assignment
- Method invocation
- Inheritance

**Not-so-fundamental issues (that we'll skip):**
- Dot expression syntax
Implementing the Object Abstraction

**Fundamental OOP concepts:**
- Object instantiation and initialization
- Attribute look-up and assignment
- Method invocation
- Inheritance

**Not-so-fundamental issues (that we'll skip):**
- Dot expression syntax
- Multiple inheritance (on your homework)
Implementing the Object Abstraction

**Fundamental OOP concepts:**
- Object instantiation and initialization
- Attribute look-up and assignment
- Method invocation
- Inheritance

**Not-so-fundamental issues (that we'll skip):**
- Dot expression syntax
- Multiple inheritance (on your homework)
- Introspection (e.g., what class does this object have?)
Instances
Instances

Dispatch dictionary with messages 'get' and 'set'
Instances

Dispatch dictionary with messages 'get' and 'set'

Attributes stored in a local dictionary "attributes"
Instances

Dispatch dictionary with messages 'get' and 'set'

Attributes stored in a local dictionary "attributes"
Instances

Dispatch dictionary with messages 'get' and 'set'

Attributes stored in a local dictionary "attributes"

```python
def make_instance(cls):
    """Return a new object instance."""

def get_value(name):
    if name in attributes:
        return attributes[name]
    else:
        value = cls['get'](name)
        return bind_method(value, instance)

def set_value(name, value):
    attributes[name] = value

attributes = {}
instance = {'get': get_value, 'set': set_value}
return instance

(Demo)
Instances

Dispatch dictionary with messages 'get' and 'set'

Attributes stored in a local dictionary "attributes"

```python
def make_instance(cls):
    """Return a new object instance."""
    attributes = {}
    instance = {'get': get_value, 'set': set_value}
    return instance

    def get_value(name):
        if name in attributes:
            return attributes[name]
        else:
            value = cls['get'](name)
            return bind_method(value, instance)

    def set_value(name, value):
        attributes[name] = value
```

The class of the instance
Instances

Dispatch dictionary with messages 'get' and 'set'

Attributes stored in a local dictionary "attributes"

```python
def make_instance(cls):
    """Return a new object instance."""

def get_value(name):
    if name in attributes:
        return attributes[name]
    else:
        value = cls['get'](name)
        return bind_method(value, instance)

def set_value(name, value):
    attributes[name] = value

attributes = {}
instance = {'get': get_value, 'set': set_value}
return instance
```

(The Demo)
Dispatch dictionary with messages 'get' and 'set'

Attributes stored in a local dictionary "attributes"

```python
def make_instance(cls):
    """Return a new object instance."""
    attributes = {}
    instance = {'get': get_value, 'set': set_value}
    return instance

def get_value(name):
    if name in attributes:
        return attributes[name]
    else:
        value = cls['get'](name)
        return bind_method(value, instance)

def set_value(name, value):
    attributes[name] = value
```

The class of the instance

Match name against instance attributes

Look up the name in the class

(Demo)
Instances

Dispatch dictionary with messages 'get' and 'set'

Attributes stored in a local dictionary "attributes"

```python
def make_instance(cls):
    """Return a new object instance."""

def get_value(name):
    if name in attributes:
        return attributes[name]
    else:
        value = cls['get'](name)
        return bind_method(value, instance)

def set_value(name, value):
    attributes[name] = value

attributes = {}
instance = { 'get': get_value, 'set': set_value }
return instance
```

(Demo)
Bound Methods
Bound Methods

If looking up a name returns a class attribute value that is a function, getattr returns a bound method

Bound Methods

If looking up a name returns a class attribute value that is a function, getattr returns a bound method

```python
def make_instance(cls):
    def get_value(name):
        if name in attributes:
            return attributes[name]
        else:
            value = cls['get'](name)
            return bind_method(value, instance)
...
```
Bound Methods

If looking up a name returns a class attribute value that is a function, getattr returns a bound method

def make_instance(cls):
    def get_value(name):
        if name in attributes:
            return attributes[name]
        else:
            value = cls['get'](name)
            return bind_method(value, instance)

...
Bound Methods

If looking up a name returns a class attribute value that is a function, `getattr` returns a bound method

```python
def make_instance(cls):
    def get_value(name):
        if name in attributes:
            return attributes[name]
        else:
            value = cls['get'](name)
            return bind_method(value, instance)
    ...
```

(Demo)
Classes
Classes

Dispatch dictionaries with messages 'get', 'set', and 'new'
Classes

Dispatch dictionaries with messages 'get', 'set', and 'new'
Classes

Dispatch dictionaries with messages 'get', 'set', and 'new'

```python
def make_class(attributes={}, base_class=None):
    """Return a new class."""

    def get_value(name):
        if name in attributes:
            return attributes[name]
        elif base_class is not None:
            return base_class['get'](name)

    def set_value(name, value):
        attributes[name] = value

    def new(*args):
        return init_instance(cls, *args)

    cls = {'get': get_value, 'set': set_value, 'new': new}
    return cls

(Demo)```
Classes

Dispatch dictionaries with messages 'get', 'set', and 'new'

```python
def make_class(attributes={}, base_class=None):
    """Return a new class."""
    def get_value(name):
        if name in attributes:
            return attributes[name]
        elif base_class is not None:
            return base_class['get'](name)

def set_value(name, value):
    attributes[name] = value

def new(*args):
    return init_instance(cls, *args)

cls = {'get': get_value, 'set': set_value, 'new': new}
return cls
```

(Demo)
Classes

Dispatch dictionaries with messages 'get', 'set', and 'new'

```python
def make_class(attributes={}, base_class=None):
    """Return a new class."""
    def get_value(name):
        if name in attributes:
            return attributes[name]
        elif base_class is not None:
            return base_class['get'](name)
    def set_value(name, value):
        attributes[name] = value
    def new(*args):
        return init_instance(cls, *args)

    cls = {'get': get_value, 'set': set_value, 'new': new}
    return cls
```

(Demo)
Classes

Dispatch dictionaries with messages 'get', 'set', and 'new'

```python
def make_class(attributes={}, base_class=None):
    """Return a new class."""
    def get_value(name):
        if name in attributes:
            return attributes[name]
        elif base_class is not None:
            return base_class['get'](name)
    def set_value(name, value):
        attributes[name] = value
    def new(*args):
        return init_instance(cls, *args)
    cls = {'get': get_value, 'set': set_value, 'new': new}
    return cls
```

The class attribute look-up procedure

Common dispatch dictionary pattern

(Demo)
Instantiation and Initialization
Instantiation and Initialization

First makes a new instance, then invokes the __init__ method
Instantiation and Initialization

First makes a new instance, then invokes the `__init__` method

```python
def make_class(attributes={}, base_class=None):
    ...
    def new(*args):
        return init_instance(cls, *args)
    ...
```
Instantiation and Initialization

First makes a new instance, then invokes the __init__ method

```python
def make_class(attributes={}, base_class=None):
    ...
    def new(*args):
        return init_instance(cls, *args)
    ...

def init_instance(cls, *args):
```


Instantiation and Initialization

First makes a new instance, then invokes the `__init__` method

```python
def make_class(attributes={}, base_class=None):
    ...
    def new(*args):
        return init_instance(cls, *args)
    ...

def init_instance(cls, *args):
    """Return a new instance of cls, initialized with args."""
```
Instantiation and Initialization

First makes a new instance, then invokes the __init__ method

```python
def make_class(attributes={}, base_class=None):
    ...
    def new(*args):
        return init_instance(cls, *args)
    ...

def init_instance(cls, *args):
    """Return a new instance of cls, initialized with args."""
    instance = make_instance(cls)
```

```
Instantiation and Initialization

First makes a new instance, then invokes the `__init__` method

```python
def make_class(attributes={}, base_class=None):
    ...
    def new(*args):
        return init_instance(cls, *args)
    ...

def init_instance(cls, *args):
    """Return a new instance of cls, initialized with args.""
    instance = make_instance(cls)
```

Dispatch dictionary
Instantiation and Initialization

First makes a new instance, then invokes the __init__ method

```python
def make_class(attributes={}, base_class=None):
    ...
def new(*args):
    return init_instance(cls, *args)
    ...

def init_instance(cls, *args):
    """Return a new instance of cls, initialized with args.""
    instance = make_instance(cls)
    init = cls['get']('__init__')
```

Dispatch dictionary
Instantiation and Initialization

First makes a new instance, then invokes the __init__ method

```python
def make_class(attributes={}, base_class=None):
    ...
def new(*args):
    return init_instance(cls, *args)
    ...

def init_instance(cls, *args):
    """Return a new instance of cls, initialized with args.""
    instance = make_instance(cls)
    init = cls['get']('_init_')
```

Dispatch dictionary

The constructor name is fixed here
Instantiation and Initialization

First makes a new instance, then invokes the `__init__` method

```python
def make_class(attributes={}, base_class=None):
    ...
    def new(*args):
        return init_instance(cls, *args)
    ...

def init_instance(cls, *args):
    """Return a new instance of cls, initialized with args.""
    instance = make_instance(cls)
    init = cls[\'get\'](\'__init__\')
    if init is not None:
        Dispatch dictionary
        The constructor name is fixed here
```
Instantiation and Initialization

First makes a new instance, then invokes the `__init__` method

```python
def make_class(attributes={}, base_class=None):
  ...
  def new(*args):
    return init_instance(cls, *args)
  ...

def init_instance(cls, *args):
  """Return a new instance of cls, initialized with args."""
  instance = make_instance(cls)
  init = cls['get']['__init__']
  if init is not None:
    init(instance, *args)
```

Dispatch dictionary

The constructor name is fixed here
Instantiation and Initialization

First makes a new instance, then invokes the __init__ method

```python
def make_class(attributes={}, base_class=None):
    ...
    def new(*args):
        return init_instance(cls, *args)
    ...

def init_instance(cls, *args):
    """Return a new instance of cls, initialized with args.""
    instance = make_instance(cls);  # Dispatch dictionary
    init = cls['get']('__init__')
    if init is not None:
        init(instance, *args)
    return instance
```

The constructor name is fixed here
Example: Defining an Account Class
Example: Defining an Account Class

(Demo)
Example: Defining an Account Class

```python
def make_account_class():  # (Demo)

    interest = 0.02

    def __init__(self, account_holder):
        self['set']['holder', account_holder]
        self['set']['balance', 0]

    def deposit(self, amount):
        new_balance = self['get']['balance'] + amount
        self['set']['balance', new_balance]
        return self['get']['balance']

    def withdraw(self, amount):
        balance = self['get']['balance']
        if amount > balance:
            return 'Insufficient funds'
        self['set']['balance', balance - amount]
        return self['get']['balance']

    return make_class(locals())

Account = make_account_class()
```
Example: Defining an Account Class

```python
def make_account_class():  # (Demo)
    interest = 0.02

    def __init__(self, account_holder):
        self['set']('holder', account_holder)
        self['set']('balance', 0)

    def deposit(self, amount):
        new_balance = self['get']('balance') + amount
        self['set']('balance', new_balance)
        return self['get']('balance')

    def withdraw(self, amount):
        balance = self['get']('balance')
        if amount > balance:
            return 'Insufficient funds'
        self['set']('balance', balance - amount)
        return self['get']('balance')

    return make_class(locals())

Account = make_account_class()
```
Example: Using the Account Class

The Account class is instantiated and stored, then messaged
Example: Using the Account Class

The Account class is instantiated and stored, then messaged

```python
>>> Account = make_account_class()
```
Example: Using the Account Class

The Account class is instantiated and stored, then messaged

```python
>>> Account = make_account_class()
>>> jim_acct = Account['new']('Jim')
```
Example: Using the Account Class

The Account class is instantiated and stored, then messaged

```python
>>> Account = make_account_class()
>>> jim_acct = Account['new']('Jim')
>>> jim_acct['get']('holder')
'Jim'
```
Example: Using the Account Class

The Account class is instantiated and stored, then messaged

```python
>>> Account = make_account_class()
>>> jim_acct = Account['new']('Jim')
>>> jim_acct['get']('holder')
'Jim'
>>> jim_acct['get']('interest')
0.02
```
Example: Using the Account Class

The Account class is instantiated and stored, then messaged

```python
>>> Account = make_account_class()
>>> jim_acct = Account['new']('Jim')
>>> jim_acct['get']('holder')
'Jim'
>>> jim_acct['get']('interest')
0.02
>>> jim_acct['get']('deposit')(20)
20
```
Example: Using the Account Class

The Account class is instantiated and stored, then messaged

```python
>>> Account = make_account_class()
>>> jim_acct = Account['new']('Jim')
>>> jim_acct['get']('holder')
'Jim'
>>> jim_acct['get']('interest')
0.02
>>> jim_acct['get']('deposit')(20)
20
>>> jim_acct['get']('withdraw')(5)
15
```
Example: Using the Account Class

The Account class is instantiated and stored, then messaged

```python
>>> Account = make_account_class()
>>> jim_acct = Account['new']('Jim')
>>> jim_acct['get']('holder')
'Jim'
>>> jim_acct['get']('interest')
0.02
>>> jim_acct['get']('deposit')(20)
20
>>> jim_acct['get']('withdraw')(5)
15
```

How can we also use getattr and setattr style syntax?
Class and Instance Attributes

Instance attributes and class attributes can share names
Instance attributes and class attributes can share names

```python
>>> Account = make_account_class()
```
Class and Instance Attributes

Instance attributes and class attributes can share names

```python
>>> Account = make_account_class()
>>> jim_acct = Account['new']('Jim')
```
Class and Instance Attributes

Instance attributes and class attributes can share names

```python
>>> Account = make_account_class()
>>> jim_acct = Account['new']('Jim')
>>> jim_acct['set']('interest', 0.08)
```
Class and Instance Attributes

Instance attributes and class attributes can share names

```python
>>> Account = make_account_class()
>>> jim_acct = Account['new']('Jim')
>>> jim_acct['set']( 'interest', 0.08)
>>> Account['get']( 'interest')
0.02
```
Class and Instance Attributes

Instance attributes and class attributes can share names

```python
>>> Account = make_account_class()
>>> jim_acct = Account['new']('Jim')
>>> jim_acct['set']('interest', 0.08)
>>> Account['get']('interest')
0.02
```

(Demo)
Example: Using Inheritance

CheckingAccount is a special case of Account
Example: Using Inheritance

CheckingAccount is a special case of Account

(Demo)
Example: Using Inheritance

CheckingAccount is a special case of Account

def make_checking_account_class():

    interest = 0.01
    withdraw_fee = 1

    def withdraw(self, amount):
        fee = self['get']('withdraw_fee')
        return Account['get']('withdraw')(self, amount + fee)

    return make_class(locals(), Account)

CheckingAccount = make_checking_account_class()
Example: Using Inheritance

CheckingAccount is a special case of Account

def make_checking_account_class():
    interest = 0.01
    withdraw_fee = 1

    def withdraw(self, amount):
        fee = self['get']('withdraw_fee')
        return Account['get']('withdraw')(self, amount + fee)

    return make_class(locals(), Account)

CheckingAccount = make_checking_account_class()

(Demo)
Relationship to the Python Object System

Object attributes are stored as dictionaries

Some "magic" names, __<name>__, require special handling

An object has an "attribute" called __dict__ that is a dictionary of its user-defined instance attributes

(Demo)
Relationship to the Python Object System

Object attributes are stored as dictionaries

Some "magic" names, __<name>__, require special handling

An object has an "attribute" called __dict__ that is a dictionary of its user-defined instance attributes

(Demo)

In Python, classes have classes too

The equivalent of init_instance can be customized (metaclass)