Object-Oriented Programming

Slides adapted from Berkeley cs61a

- OOP
- Classes and Objects
- Methods and Attributes
- Lookup up Attributes by Name

OOP, an example

Object-Oriented Programming

A method for organizing programs

- Data abstraction
- Bundling together information and related behavior

Object-Oriented Programming

A method for organizing programs

- Data abstraction
- Bundling together information and related behavior

- A metaphor for computation using distributed state
- Each object has its own local state
- Each object also knows how to manage its own local state, based on method calls
- Method calls are messages passed between objects
- Several objects may all be instances of a common type
- Different types may relate to each other

Object-Oriented Programming

A method for organizing programs

- Data abstraction
- Bundling together information and related behavior

- A metaphor for computation using distributed state
- Each object has its own local state
- Each object also knows how to manage its own local state, based on method calls
- Method calls are messages passed between objects
- Several objects may all be instances of a common type
- Different types may relate to each other

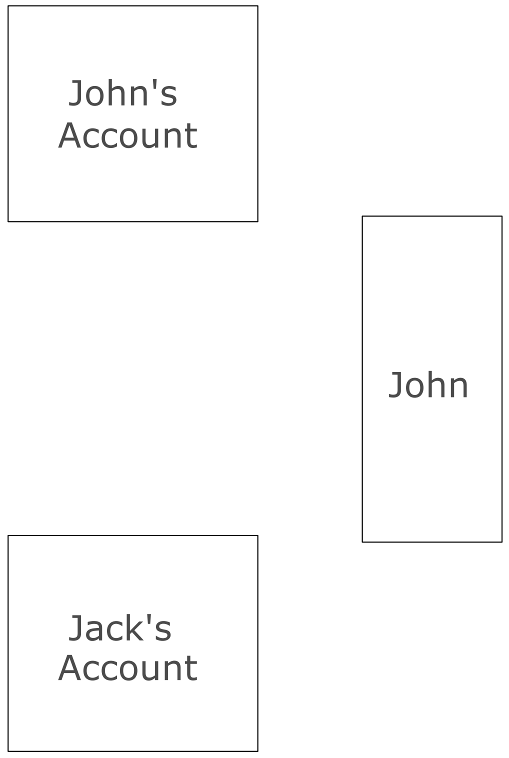
Specialized syntax & vocabulary to support this metaphor

Object-Oriented Programming

A method for organizing programs

- Data abstraction
- Bundling together information and related behavior

- A metaphor for computation using distributed state
- Each object has its own local state
- Each object also knows how to manage its own local state, based on method calls
- Method calls are messages passed between objects
- Several objects may all be instances of a common type
- Different types may relate to each other



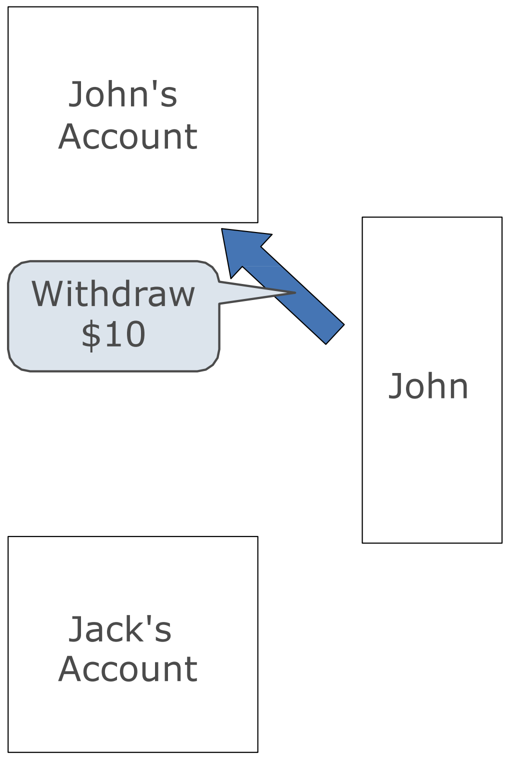
Specialized syntax & vocabulary to support this metaphor

Object-Oriented Programming

A method for organizing programs

- Data abstraction
- Bundling together information and related behavior

- A metaphor for computation using distributed state
- Each object has its own local state
- Each object also knows how to manage its own local state, based on method calls
- Method calls are messages passed between objects
- Several objects may all be instances of a common type
- Different types may relate to each other



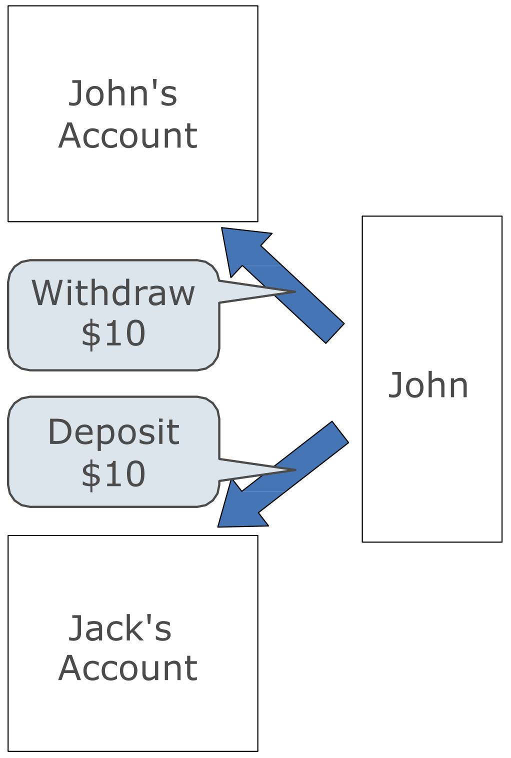
Specialized syntax & vocabulary to support this metaphor

Object-Oriented Programming

A method for organizing programs

- Data abstraction
- Bundling together information and related behavior

- A metaphor for computation using distributed state
- Each object has its own local state
- Each object also knows how to manage its own local state, based on method calls
- Method calls are messages passed between objects
- Several objects may all be instances of a common type
- Different types may relate to each other

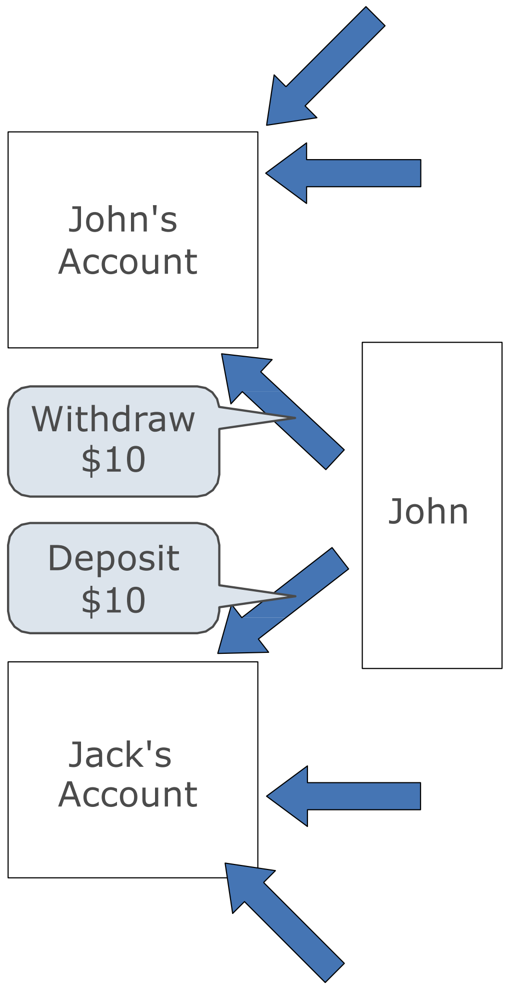


Specialized syntax & vocabulary to support this metaphor

Object-Oriented Programming

A method for organizing programs

- Data abstraction
- Bundling together information and related behavior
- A metaphor for computation using distributed state
- Each object has its own local state
- Each object also knows how to manage its own local state, based on method calls
- Method calls are messages passed between objects
- Several objects may all be instances of a common type
- Different types may relate to each other



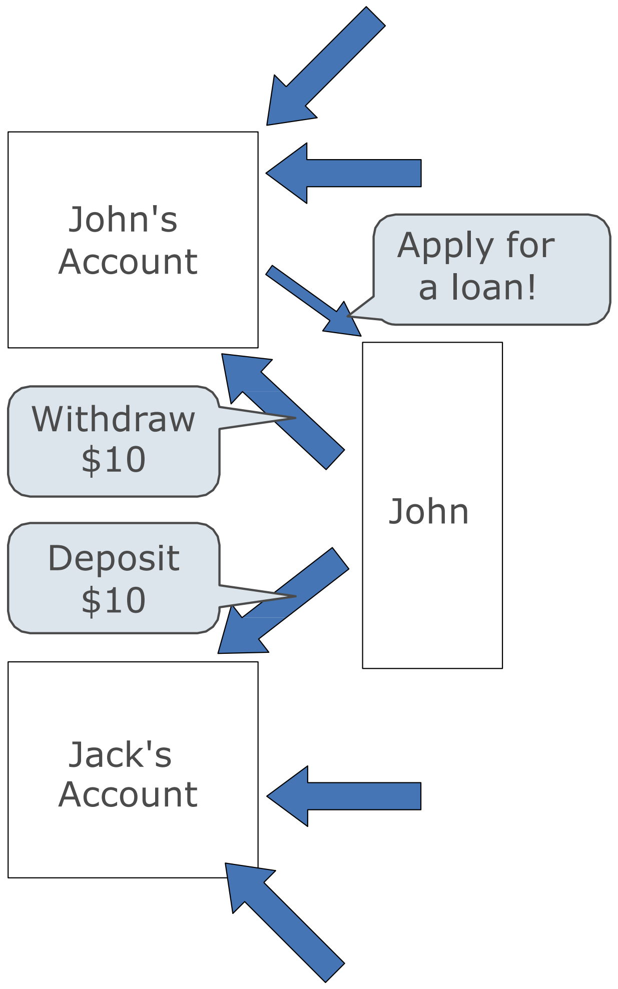
Specialized syntax & vocabulary to support this metaphor

Object-Oriented Programming

A method for organizing programs

- Data abstraction
- Bundling together information and related behavior

- A metaphor for computation using distributed state
- Each object has its own local state
- Each object also knows how to manage its own local state, based on method calls
- Method calls are messages passed between objects
- Several objects may all be instances of a common type
- Different types may relate to each other



Specialized syntax & vocabulary to support this metaphor

Classes

A class serves as a template for its instances

Classes

A class serves as a template for its instances

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance

Classes

A class serves as a template for its instances

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance

```
>>> a = Account ('John')
```

Classes

A class serves as a template for its instances

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance

```
>>> a = Account ('John')
>>> a.holder
'John'
```

Classes

A class serves as a template for its instances

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance

```
>>> a = Account ('John')
>>> a.holder
'John'
>>> a.balance
0
```

Classes

A class serves as a template for its instances

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance

```
>>> a = Account('John')
>>> a.holder
'John'
>>> a.balance
0
```

Idea: All bank accounts should have withdraw and deposit behaviors that all work in the same way

Classes

A class serves as a template for its instances

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance

```
>>> a = Account ('John')
>>> a.holder
'John'
>>> a.balance
0
```

Idea: All bank accounts should have withdraw and deposit behaviors that all work in the same way

```
>>> a.deposit(15)
15
```

Classes

A class serves as a template for its instances

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance

```
>>> a = Account ('John')
>>> a.holder
'John'
>>> a.balance
0
```

Idea: All bank accounts should have withdraw and deposit behaviors that all work in the same way

```
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
```

Classes

A class serves as a template for its instances

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance

```
>>> a = Account ('John')
>>> a.holder
'John'
>>> a.balance
0
```

Idea: All bank accounts should have withdraw and deposit behaviors that all work in the same way

```
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.balance
5
```

Classes

A class serves as a template for its instances

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance

```
>>> a = Account('John')
>>> a.holder
'John'
>>> a.balance
0
```

Idea: All bank accounts should have withdraw and deposit behaviors that all work in the same way

```
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.balance
5
>>> a.withdraw(10)
'Insufficient funds'
```

Classes

A class serves as a template for its instances

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance

```
>>> a = Account ('John')
>>> a.holder
'John'
>>> a.balance
0
```

Idea: All bank accounts should have withdraw and deposit behaviors that all work in the same way

```
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.balance
5
>>> a.withdraw(10)
'Insufficient funds'
```

Better idea: All bank accounts share a withdraw method and a deposit method

Class Statements

The Class Statement

```
class <name>:
    <suite>
```

The Class Statement

```
class <name>:
    <suite>
```

A class statement creates a new class and binds that class to <name> in the first frame of the current environment

The Class Statement

```
class <name>:
    <suite>
```

A class statement creates a new class and binds that class to <name> in the first frame of the current environment

Assignment & def statements in <suite> create attributes of the class (not names in frames)

The Class Statement

```
class <name>:
    <suite>
```

The suite is executed when the class statement is executed.

A class statement creates a new class and binds that class to <name> in the first frame of the current environment

Assignment & def statements in <suite> create attributes of the class (not names in frames)

The Class Statement

```
class <name>:
    <suite>
```

The suite is executed when the class statement is executed.

A class statement creates a new class and binds that class to <name> in the first frame of the current environment

Assignment & def statements in <suite> create attributes of the class (not names in frames)

```
>>> class Clown:
...     nose = 'big and red'
...     def dance():
...         return 'No thanks'
```

The Class Statement

```
class <name>:
    <suite>
```

The suite is executed when the class statement is executed.

A class statement creates a new class and binds that class to <name> in the first frame of the current environment

Assignment & def statements in <suite> create attributes of the class (not names in frames)

```
>>> class Clown:
...     nose = 'big and red'
...     def dance():
...         return 'No thanks'

>>> Clown.nose
'big and red'
```

The Class Statement

```
class <name>:
    <suite>
```

The suite is executed when the class statement is executed.

A class statement creates a new class and binds that class to <name> in the first frame of the current environment

Assignment & def statements in <suite> create attributes of the class (not names in frames)

```
>>> class Clown:
...     nose = 'big and red'
...     def dance():
...         return 'No thanks'

>>> Clown.nose
'big and red'
>>> Clown.dance()
'No thanks'
```

The Class Statement

```
class <name>:
    <suite>
```

The suite is executed when the class statement is executed.

A class statement creates a new class and binds that class to <name> in the first frame of the current environment

Assignment & def statements in <suite> create attributes of the class (not names in frames)

```
>>> class Clown:
...     nose = 'big and red'
...     def dance():
...         return 'No thanks'

>>> Clown.nose
'big and red'
>>> Clown.dance()
'No thanks'
>>> Clown
<class '__main__.Clown'>
```

Object Construction

Idea: All bank accounts have a **balance** and an account **holder**;
the **Account** class should add those attributes to each of its instances

```
>>> a = Account('Jim')
```

Object Construction

Idea: All bank accounts have a **balance** and an account **holder**;
the **Account** class should add those attributes to each of its instances

```
>>> a = Account('Jim')
```

When a class is called:

Object Construction

Idea: All bank accounts have a **balance** and an account **holder**;
the **Account** class should add those attributes to each of its instances

```
>>> a = Account('Jim')
```

When a class is called:

1. A new instance of that class is created:

Object Construction

Idea: All bank accounts have a **balance** and an account **holder**;
the **Account** class should add those attributes to each of its instances

```
>>> a = Account('Jim')
```

When a class is called:

1. A new instance of that class is created:

Object Construction

Idea: All bank accounts have a **balance** and an account **holder**;
the **Account** class should add those attributes to each of its instances

```
>>> a = Account('Jim')
```

When a class is called:

1. A new instance of that class is created:

2. The __init__ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression

Object Construction

Idea: All bank accounts have a **balance** and an account **holder**;
the **Account** class should add those attributes to each of its instances

```
>>> a = Account('Jim')
```

When a class is called:

1. A new instance of that class is created:

2. The __init__ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression

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

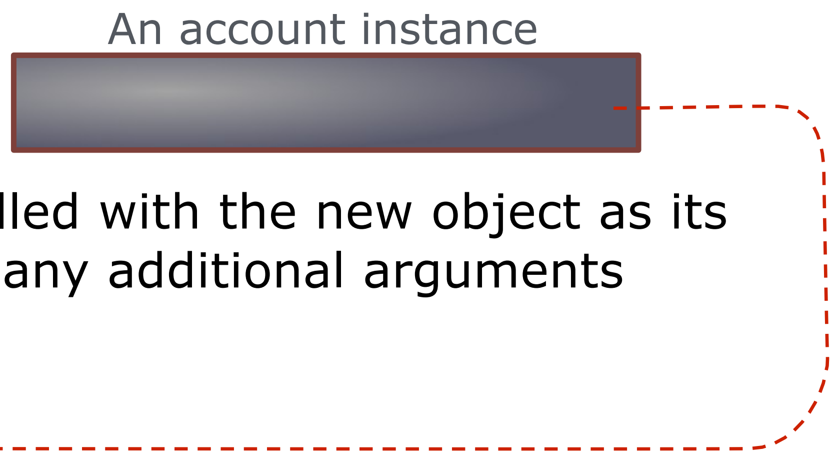
Object Construction

Idea: All bank accounts have a **balance** and an account **holder**;
the **Account** class should add those attributes to each of its instances

```
>>> a = Account('Jim')
```

When a class is called:

1. A new instance of that class is created:



2. The __init__ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression

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

Object Construction

Idea: All bank accounts have a **balance** and an account **holder**;
the **Account** class should add those attributes to each of its instances

```
>>> a = Account('Jim')
```

When a class is called:

1. A new instance of that class is created:

An account instance

2. The __init__ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression

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

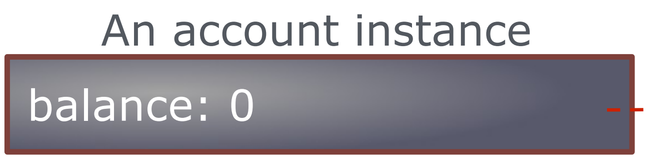
Object Construction

Idea: All bank accounts have a **balance** and an account **holder**;
the **Account** class should add those attributes to each of its instances

```
>>> a = Account('Jim')
```

When a class is called:

1. A new instance of that class is created:



2. The __init__ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression

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

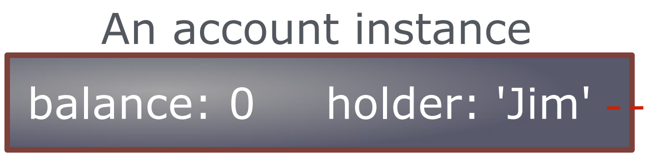
Object Construction

Idea: All bank accounts have a **balance** and an account **holder**; the **Account** class should add those attributes to each of its instances

```
>>> a = Account('Jim')
```

When a class is called:

1. A new instance of that class is created:



2. The __init__ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression

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

Object Construction

Idea: All bank accounts have a **balance** and an account **holder**;
the **Account** class should add those attributes to each of its instances

```
>>> a = Account('Jim')
```

When a class is called:

1. A new instance of that class is created:

An account instance

balance: 0	holder: 'Jim'

2. The __init__ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression

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

__init__ is called a constructor

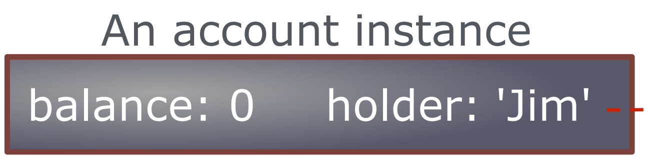
Object Construction

Idea: All bank accounts have a **balance** and an account **holder**; the **Account** class should add those attributes to each of its instances

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
```

When a class is called:

1. A new instance of that class is created:



2. The __init__ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression

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

__init__ is called a constructor

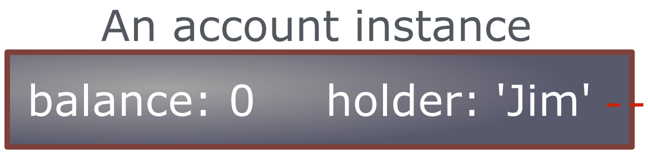
Object Construction

Idea: All bank accounts have a **balance** and an account **holder**; the **Account** class should add those attributes to each of its instances

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

When a class is called:

1. A new instance of that class is created:



2. The __init__ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression

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

__init__ is called a constructor

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('John')
>>> b = Account('Jack')
```

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('John')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('John')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Every call to Account creates a new Account instance. There is only one Account class.

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('John')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Every call to Account creates a new Account instance. There is only one Account class.

Identity operators "is" and "is not" test if two expressions evaluate to the same object:

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('John')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Every call to Account creates a new Account instance. There is only one Account class.

Identity operators "is" and "is not" test if two expressions evaluate to the same object:

```
>>> a is a
True
```

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('John')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Every call to Account creates a new Account instance. There is only one Account class.

Identity operators "is" and "is not" test if two expressions evaluate to the same object:

```
>>> a is a
True
>>> a is not b
```

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('John')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Every call to Account creates a new Account instance. There is only one Account class.

Identity operators "is" and "is not" test if two expressions evaluate to the same object:

```
>>> a is a
True
>>> a is not b
True
```

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('John')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Every call to Account creates a new Account instance. There is only one Account class.

Identity operators "is" and "is not" test if two expressions evaluate to the same object:

```
>>> a is a
True
>>> a is not b
True
```

Binding an object to a new name using assignment does not create a new object:

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('John')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Every call to Account creates a new Account instance. There is only one Account class.

Identity operators "is" and "is not" test if two expressions evaluate to the same object:

```
>>> a is a
True
>>> a is not b
True
```

Binding an object to a new name using assignment does not create a new object:

```
>>> c = a
>>> c is a
```

Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```
>>> a = Account('John')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Every call to Account creates a new Account instance. There is only one Account class.

Identity operators "is" and "is not" test if two expressions evaluate to the same object:

```
>>> a is a
True
>>> a is not b
True
```

Binding an object to a new name using assignment does not create a new object:

```
>>> c = a
>>> c is a
True
```

Methods

Methods

Methods are functions defined in the suite of a class statement

```
class Account:
```

Methods

Methods are functions defined in the suite of a class statement

```
class Account:

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

Methods

Methods are functions defined in the suite of a class statement

```
class Account:

    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    def deposit(self, amount):
```

self should always be bound to an instance of the Account class

Methods

Methods are functions defined in the suite of a class statement

```
class Account:

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

self should always be bound to an instance of the Account class

```
    def deposit(self, amount):
        self.balance = self.balance + amount
```

Methods

Methods are functions defined in the suite of a class statement

```
class Account:

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

self should always be bound to an instance of the Account class

```
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Methods

Methods are functions defined in the suite of a class statement

```
class Account:

    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
```

self should always be bound to an instance of the Account class

Methods

Methods are functions defined in the suite of a class statement

```
class Account:

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

self should always be bound to an instance of the Account class

```
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        if amount > self.balance:
```

Methods

Methods are functions defined in the suite of a class statement

```
class Account:

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

self should always be bound to an instance of the Account class

```
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds’
```

Methods

Methods are functions defined in the suite of a class statement

```
class Account:

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

self should always be bound to an instance of the Account class

```
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds’
        self.balance = self.balance - amount
```

Methods

Methods are functions defined in the suite of a class statement

```
class Account:

    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

self should always be bound to an instance of the Account class

Methods

Methods are functions defined in the suite of a class statement

```
class Account:

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

self should always be bound to an instance of the Account class

```
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

These def statements create function objects as always, but their names are bound as attributes of the class (not bound to the particular frame)

Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state

Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state

```
class Account:
    . . .
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Defined with two parameters

Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state

Defined with two parameters

```
class Account:
    . . .
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Dot notation automatically supplies the first argument to a method

Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state

Defined with two parameters

```
class Account:
    . . .
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Dot notation automatically supplies the first argument to a method

```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
```

Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state

Defined with two parameters

```
class Account:
    . . .
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Dot notation automatically supplies the first argument to a method

```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
```

Invoked with one argument

Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state

```
class Account:
    . . .
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Defined with two parameters

Dot notation automatically supplies the first argument to a method

```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
```

Bound to self

Invoked with one argument

Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state

Defined with two parameters

```
class Account:
    . . .
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Dot notation automatically supplies the first argument to a method

```
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
```

Bound to self

Invoked with one argument

(demo_1)

Attributes

Class Attributes

Class attributes are "shared" across all instances of a class because they are attributes of the class, not the instance (**Instance attributes?**)

```
class Account:

    interest = 0.02 # A class attribute

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

Class Attributes

Class attributes are "shared" across all instances of a class
because they are attributes of the class, not the instance (**Instance attributes?**)

```
class Account:

    interest = 0.02 # A class attribute

    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

>>> tom_account = Account('Tom')
>>> jim_account = Account('Jim')
```

Class Attributes

Class attributes are "shared" across all instances of a class because they are attributes of the class, not the instance (**Instance attributes?**)

```
class Account:

    interest = 0.02 # A class attribute

    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

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

Class Attributes

Class attributes are "shared" across all instances of a class
because they are attributes of the class, not the instance (**Instance attributes?**)

```
class Account:

    interest = 0.02 # A class attribute

    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

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

The **interest** attribute is ***not*** part of the instance; it's part of the class!

Class Attributes

Class attributes are "shared" across all instances of a class
because they are attributes of the class, not the instance (**Instance attributes?**)

```
class Account:

    interest = 0.02 # A class attribute

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

Methods are also considered as the attributes of the class

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

The **interest** attribute is ***not*** part of the instance; it's part of the class!

Accessing Attributes

Using getattr, we can look up an attribute using a string

```
>>> getattr(tom_account, 'balance')
10
>>> hasattr(tom_account, 'deposit')
True
```

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

(We will examine this in details later)

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

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

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

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

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

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

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

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

>>> Account.deposit(tom_account, 1000)
1000
```

Function: all arguments within parentheses

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

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

>>> Account.deposit(tom_account, 1000)
1000
>>> tom_account.deposit(1021)
2021
```

Function: all arguments within parentheses

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

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

```
>>> type(tom_account)
<class '__main__.Account'>

>>> type(Account)
<class 'type’>
```

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

```
>>> type(tom_account)
<class '__main__.Account'>

>>> type(Account)
<class 'type’>
```

?

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

We define class to define objects:
type(my_object) -> MyClass

```
>>> type(tom_account)
<class '__main__.Account'>

>>> type(Account)
<class 'type'>
```

?

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

```
>>> type(tom_account)
<class '__main__.Account'>

>>> type(Account)
<class 'type’>
```

?

We define class to define objects:
type(my_object) -> MyClass

As classes are objects in Python,
what we use to define “class objects”?

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

```
>>> type(tom_account)
<class '__main__.Account'>

>>> type(Account)
<class 'type'>
```

?

We define class to define objects:
type(my_object) -> MyClass

As classes are objects in Python,
what we use to define "class objects"?

We use **metaclass** to define classes:
type(MyClass) -> MetaClass

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

```
>>> type(tom_account)
<class '__main__.Account'>

>>> type(Account)
<class 'type'>
```

?

We define class to define objects:
type(my_object) -> MyClass

As classes are objects in Python,
what we use to define "class objects"?

We use **metaclass** to define classes:
type(MyClass) -> MetaClass

my_object = MyClass()
MyClass = MetaClass()

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

```
>>> type(tom_account)
<class '__main__.Account'>

>>> type(Account)
<class 'type'>
```
?

We define class to define objects:
type(my_object) -> MyClass

type is the metaclass in Python

As classes are objects in Python,
what we use to define "class objects"?

We use **metaclass** to define classes:
type(MyClass) -> MetaClass

my_object = MyClass()
MyClass = MetaClass()

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

```
>>> type(tom_account)
<class '__main__.Account'>

>>> type(Account)
<class 'type’>
```

?

We define class to define objects:
type(my_object) -> MyClass

type is the metaclass in Python

ACGN = type(‘ACGN’,
    (tuple for parent classes),
    {dic for attribute pairs})
print(ACGN)
type(ACGN)

As classes are objects in Python,
what we use to define “class objects”?

We use **metaclass** to define classes:
type(MyClass) -> MetaClass

my_object = MyClass()
MyClass = MetaClass()

Terminology: Attributes, Functions, and Methods

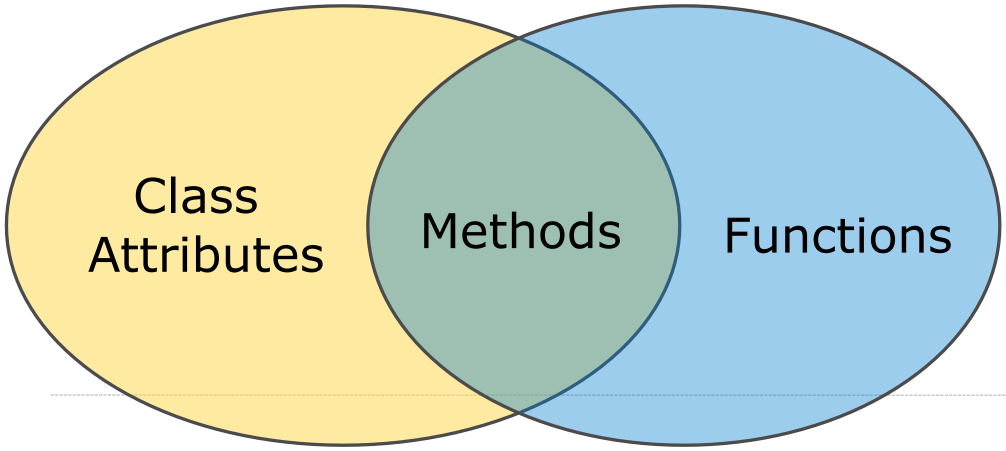
All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

Instance attribute: attribute of an instance

Class attribute: attribute of the class

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

<instance>.<method_name>

Looking Up Attributes by Name

```
<expression> . <name>
```

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

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

(demo: lls.balance)

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 (if no such class attribute exists, an AttributeError is reported)

(demo: lls.interest, lls.noSuchAttribute)

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 (if no such class attribute exists, an AttributeError is reported)
4. That value is returned unless it is a function, in which case a bound method is returned instead

(demo_2)

The X You Need To Understand In This Lecture

- The basic idea of OOP
- Classes vs. Objects

  What happens when instantiating an object from a class (object + __init__)
- Functions vs. Methods

  Understanding the 'self' keyword
- Instance attributes vs. Class attributes
- The rules for looking up attributes