61A Lecture 14

Wednesday, February 25

Announcements

Announcements

- Project 2 due Thursday 2/26 @ 11:59pm

Announcements

- Project 2 due Thursday 2/26 @ 11:59pm
  - Extra office hours on Wednesday 2/25 4pm-6pm in Bechtel (Garbarini Lounge)

Announcements

- Project 2 due Thursday 2/26 @ 11:59pm
  - Extra office hours on Wednesday 2/25 4pm–6pm in Bechtel (Garbarini Lounge)
  - Bonus point for early submission by Wednesday 2/25 @ 11:59pm!

Announcements

- Project 2 due Thursday 2/26 @ 11:59pm
  - Extra office hours on Wednesday 2/25 4pm-6pm in Bechtel (Garbarini Lounge)
  - Bonus point for early submission by Wednesday 2/25 @ 11:59pm!
- Relocated office hours on Thursday 2/26: 380 Soda (11am-3pm) & 606 Soda (3pm-6pm)

Object-Oriented Programming

Object-Oriented Programming

Object-Oriented Programming

A method for organizing programs

Object-Oriented Programming

A method for organizing programs

- Data abstraction

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

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

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

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

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

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

John's Account

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

John's Account

Steven's Account

Object-Oriented Programming

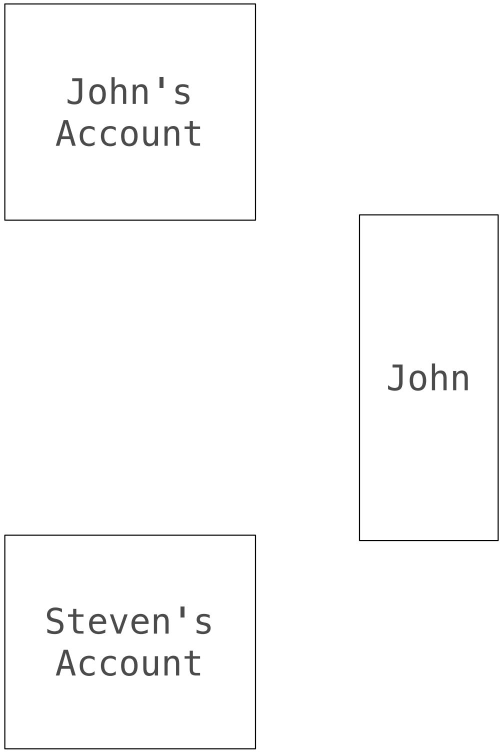
A method for organizing programs

- 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

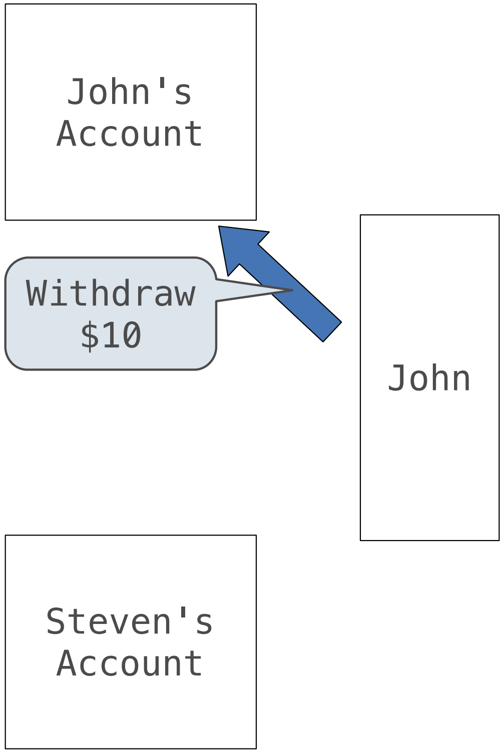
A method for organizing programs

- 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

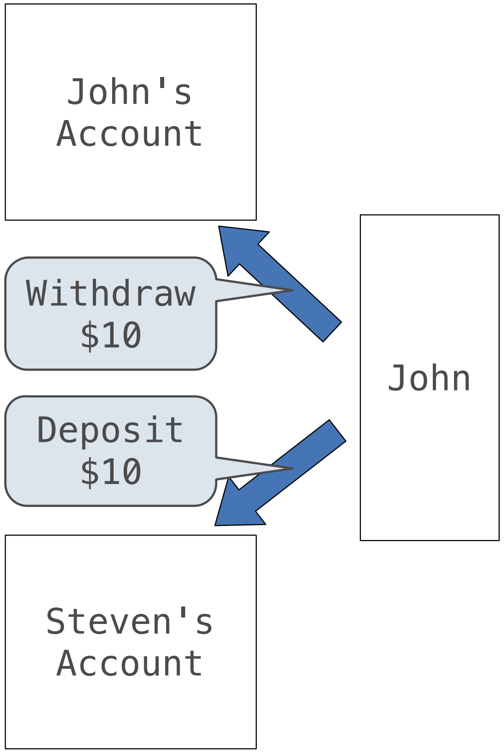
A method for organizing programs

- 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

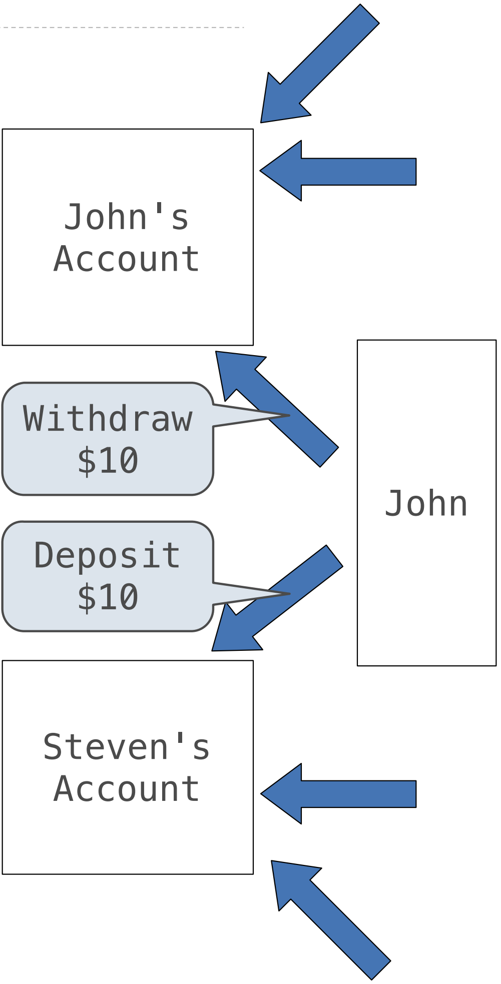
A method for organizing programs

- 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

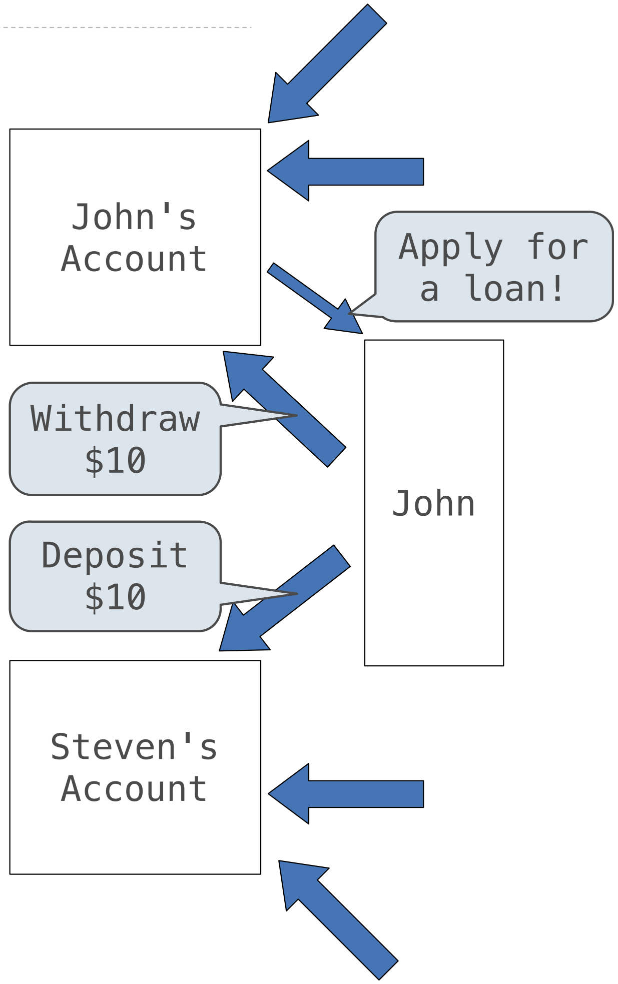
A method for organizing programs

- 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

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('Jim')
```

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('Jim')
```

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('Jim')
```

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

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('Jim')
>>> a.holder
'Jim'
>>> 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('Jim')
>>> a.holder
'Jim'
>>> 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('Jim')
>>> a.holder
'Jim'
>>> 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.

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

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

>>> 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('Jim')
>>> a.holder
'Jim'
>>> 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('Jim')
>>> a.holder
'Jim'
>>> 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.

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

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

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

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

Class Statements

The Class Statement

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

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:

An account instance

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
```

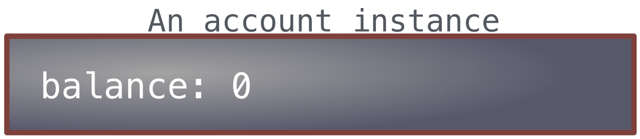
Object Construction

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:


An account instance

balance: 0

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
```

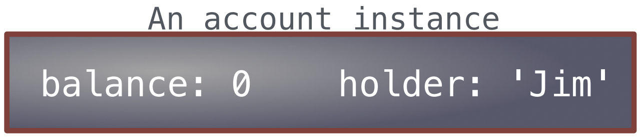
Object Construction

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

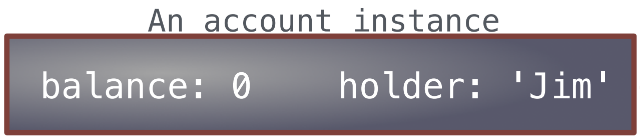
Object Construction

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

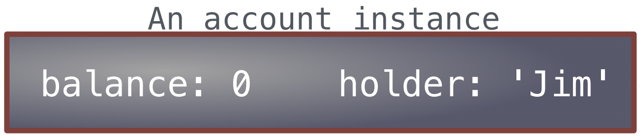
Object Construction

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:


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 Identity

Object Identity

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

Object Identity

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

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

Object Identity

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

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

Methods are functions defined in the suite of a class statement

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

```
            def __init__(self, account_holder):
```

Methods

Methods are functions defined in the suite of a class statement

```
                self.balance = 0
```

Methods

Methods are functions defined in the suite of a class statement

```
                    self.holder = account_holder
```

Methods

Methods are functions defined in the suite of a class statement

```
        def deposit(self, amount):
```

Methods

Methods are functions defined in the suite of a class statement

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

```
def deposit(self, amount):
```

Methods

Methods are functions defined in the suite of a class statement

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

```
self.balance = self.balance + amount
```

Methods

Methods are functions defined in the suite of a class statement

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

```
        return self.balance
```

Methods

Methods are functions defined in the suite of a class statement

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

```
def withdraw(self, amount):
```

Methods

Methods are functions defined in the suite of a class statement

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

```
        if amount > self.balance:
```

Methods

Methods are functions defined in the suite of a class statement

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

```
                return 'Insufficient funds'
```

Methods

Methods are functions defined in the suite of a class statement

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

```
self.balance = self.balance - amount
```

Methods

Methods are functions defined in the suite of a class statement

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

```
        return self.balance
```

Methods

Methods are functions defined in the suite of a class statement

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

```
        return self.balance
```

These def statements create function objects as always,
but their names are bound as attributes of the class

Methods

Methods are functions defined in the suite of a class statement

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

```
                return self.balance
```

These def statements create function objects as always,
but their names are bound as attributes of the class

Invoking Methods

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
```

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
```

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.

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

Dot Expressions

Dot Expressions

Objects receive messages via dot notation.

Dot Expressions

Objects receive messages via dot notation.

Dot notation accesses attributes of the instance or its class.

Dot Expressions

Objects receive messages via dot notation.

Dot notation accesses attributes of the instance or its class.

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

Dot Expressions

Objects receive messages via dot notation.

Dot notation accesses attributes of the instance or its class.

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

The <expression> can be any valid Python expression.

Dot Expressions

Objects receive messages via dot notation.

Dot notation accesses attributes of the instance or its class.

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

The <expression> can be any valid Python expression.

The <name> must be a simple name.

Dot Expressions

Objects receive messages via dot notation.

Dot notation accesses attributes of the instance or its class.

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

The <expression> can be any valid Python expression.

The <name> must be a simple name.

Evaluates to the value of the attribute looked up by <name> in the object that is the value of the <expression>.

Dot Expressions

Objects receive messages via dot notation.

Dot notation accesses attributes of the instance or its class.

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

The <expression> can be any valid Python expression.

The <name> must be a simple name.

Evaluates to the value of the attribute looked up by <name> in the object that is the value of the <expression>.

```
tom_account.deposit(10)
```

Dot Expressions

Objects receive messages via dot notation.

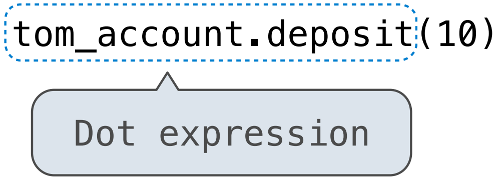
Dot notation accesses attributes of the instance or its class.

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

The <expression> can be any valid Python expression.

The <name> must be a simple name.

Evaluates to the value of the attribute looked up by <name> in the object that is the value of the <expression>.



```
tom_account.deposit(10)
```

Dot expression

Dot Expressions

Objects receive messages via dot notation.

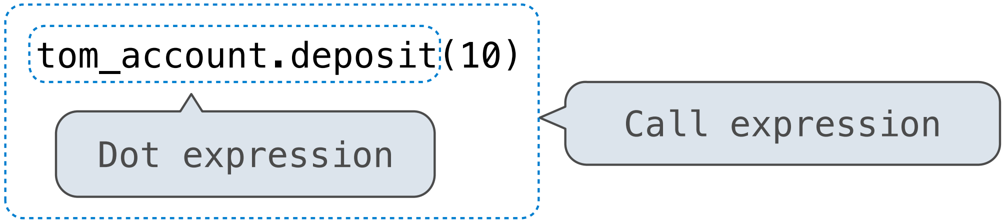
Dot notation accesses attributes of the instance or its class.

<expression> . <name>

The <expression> can be any valid Python expression.

The <name> must be a simple name.

Evaluates to the value of the attribute looked up by <name> in the object that is the value of the <expression>.



```
tom_account.deposit(10)
```

Dot expression

Call expression

Dot Expressions

Objects receive messages via dot notation.

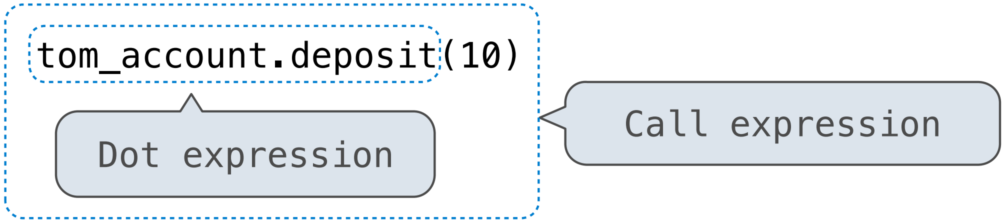
Dot notation accesses attributes of the instance or its class.

<expression> . <name>

The <expression> can be any valid Python expression.

The <name> must be a simple name.

Evaluates to the value of the attribute looked up by <name> in the object that is the value of the <expression>.



```
tom_account.deposit(10)
```

Dot expression

Call expression

(Demo)

Attributes

(Demo)

Accessing Attributes

Accessing Attributes

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

Accessing Attributes

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

```
>>> getattr(tom_account, 'balance')
10
```

Accessing Attributes

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

```
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
True
```

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

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:

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

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

Methods and Functions

Methods and Functions

Python distinguishes between:

Methods and Functions

Python distinguishes between:

- *Functions*, which we have been creating since the beginning of the course, and

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)
```

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)
```

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, 1001)
1011
```

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, 1001)
1011
>>> tom_account.deposit(1003)
2014
```

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, 1001)
1011
>>> tom_account.deposit(1003)
2014
```

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, 1001)
1011
>>> tom_account.deposit(1003)
2014
```

Function: all arguments within parentheses

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

Looking Up Attributes by Name

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

Looking Up Attributes by Name

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

To evaluate a 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.

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.

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.

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.

4. That value is returned unless it is a function, in which case a bound method is returned instead.

Class Attributes

Class Attributes

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

Class Attributes

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

```
class Account:

    interest = 0.02   # A class attribute

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

    # Additional methods would be defined here
```

Class Attributes

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

```
class Account:

    interest = 0.02   # A class attribute

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

    # Additional methods would be defined here


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

Class Attributes

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

```
class Account:

    interest = 0.02   # A class attribute

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

    # Additional methods would be defined here


>>> 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.

```
class Account:

    interest = 0.02   # A class attribute

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

    # Additional methods would be defined here
```

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

```
class Account:

    interest = 0.02   # A class attribute

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

    # Additional methods would be defined here

>>> tom_account = Account('Tom')
>>> jim_account = Account('Jim')
>>> tom_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.

```
class Account:

    interest = 0.02   # A class attribute

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

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

Attribute Assignment

Assignment to Attributes

Assignment to Attributes

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression

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

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

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

```
class Account:
    interest = 0.02
    def __init__(self, holder):
        self.holder = holder
        self.balance = 0
    ...

tom_account = Account('Tom')
```

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

```
class Account:
    interest = 0.02
    def __init__(self, holder):
        self.holder = holder
        self.balance = 0
    ...

tom_account = Account('Tom')
```

```
tom_account.interest = 0.08
```

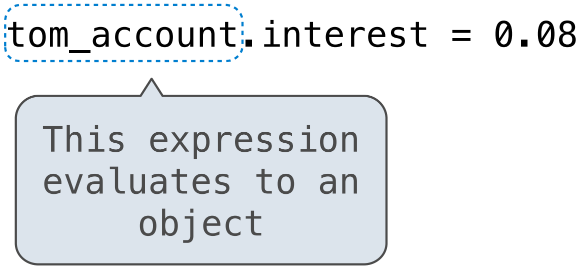
Assignment to Attributes

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

```
class Account:
    interest = 0.02
    def __init__(self, holder):
        self.holder = holder
        self.balance = 0
    ...

tom_account = Account('Tom')
```



```
tom_account.interest = 0.08
```

This expression evaluates to an object

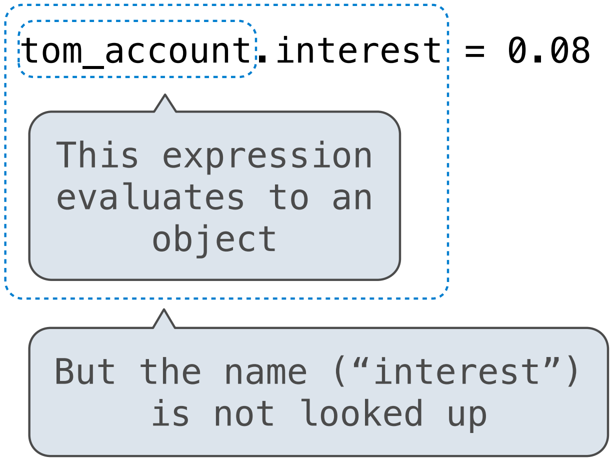
Assignment to Attributes

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

```
class Account:
    interest = 0.02
    def __init__(self, holder):
        self.holder = holder
        self.balance = 0
    ...

tom_account = Account('Tom')
```



tom_account.interest = 0.08

This expression evaluates to an object

But the name (“interest”) is not looked up

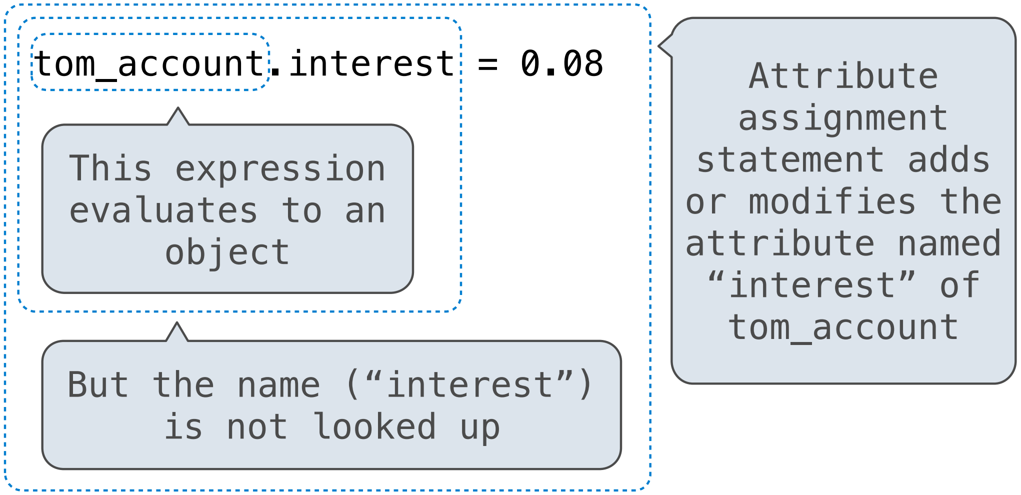
Assignment to Attributes

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

```
class Account:
    interest = 0.02
    def __init__(self, holder):
        self.holder = holder
        self.balance = 0
    ...

tom_account = Account('Tom')
```



```
tom_account.interest = 0.08
```

This expression evaluates to an object

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

Attribute assignment statement adds or modifies the attribute named "interest" of tom_account

Assignment to Attributes

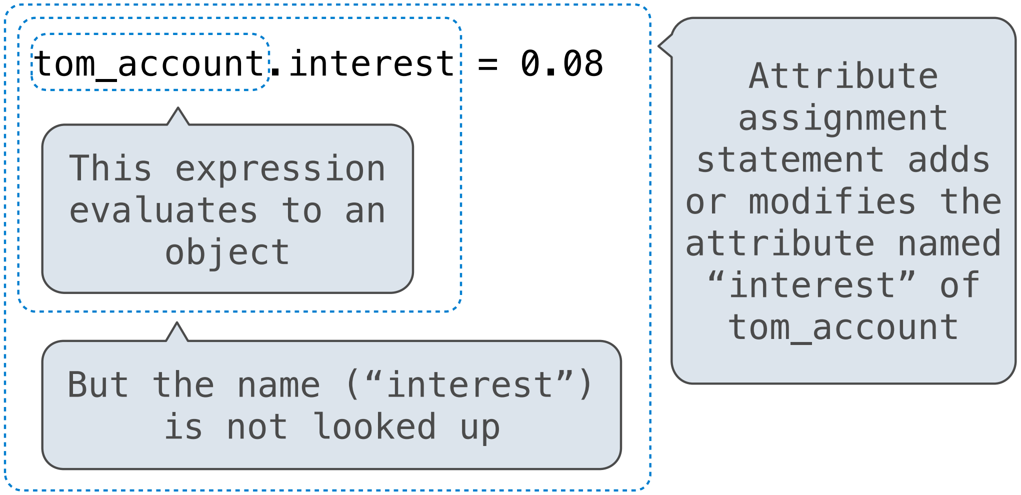
Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression

- If the object is an instance, then assignment sets an instance attribute
- If the object is a class, then assignment sets a class attribute

```
class Account:
    interest = 0.02
    def __init__(self, holder):
        self.holder = holder
        self.balance = 0
    ...

tom_account = Account('Tom')
```

Instance Attribute Assignment :



```
tom_account.interest = 0.08
```

This expression evaluates to an object

But the name (“interest”) is not looked up

Attribute assignment statement adds or modifies the attribute named “interest” of tom_account

Assignment to Attributes

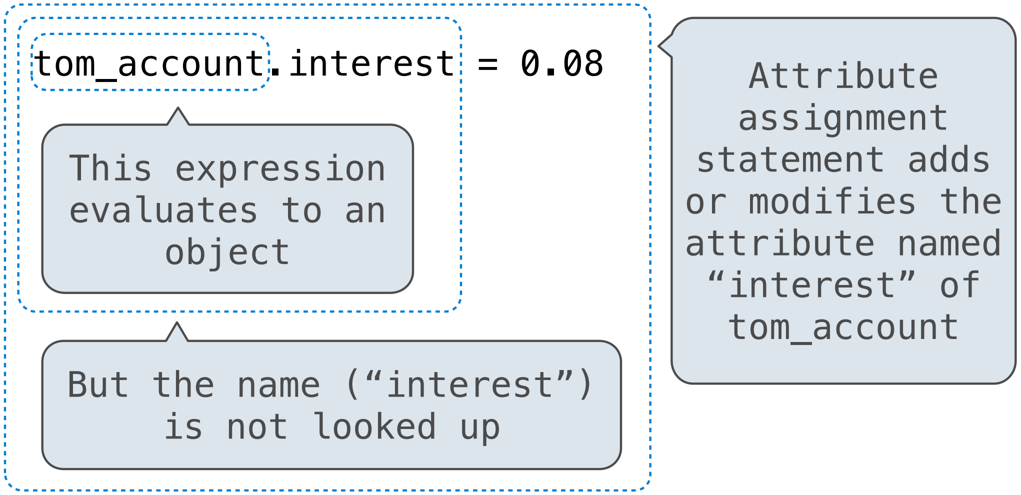
Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression

- If the object is an instance, then assignment sets an instance attribute
- If the object is a class, then assignment sets a class attribute

```
class Account:
    interest = 0.02
    def __init__(self, holder):
        self.holder = holder
        self.balance = 0
    ...

tom_account = Account('Tom')
```

Instance Attribute Assignment :



tom_account.interest = 0.08

This expression evaluates to an object

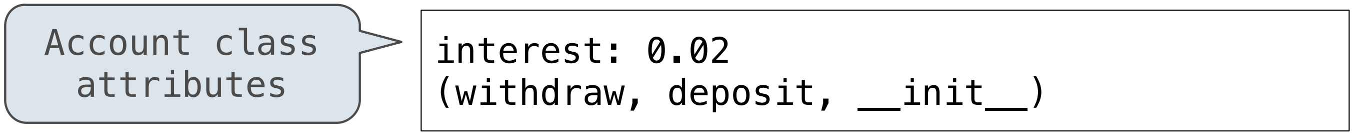
But the name (“interest”) is not looked up

Attribute assignment statement adds or modifies the attribute named “interest” of tom_account

Class Attribute Assignment :

Account.interest = 0.04

Attribute Assignment Statements



Account class attributes

```
interest: 0.02
(withdraw, deposit, __init__)
```

Attribute Assignment Statements

Account class attributes

```
interest: 0.02
(withdraw, deposit, __init__)
```

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

Attribute Assignment Statements

Account class attributes

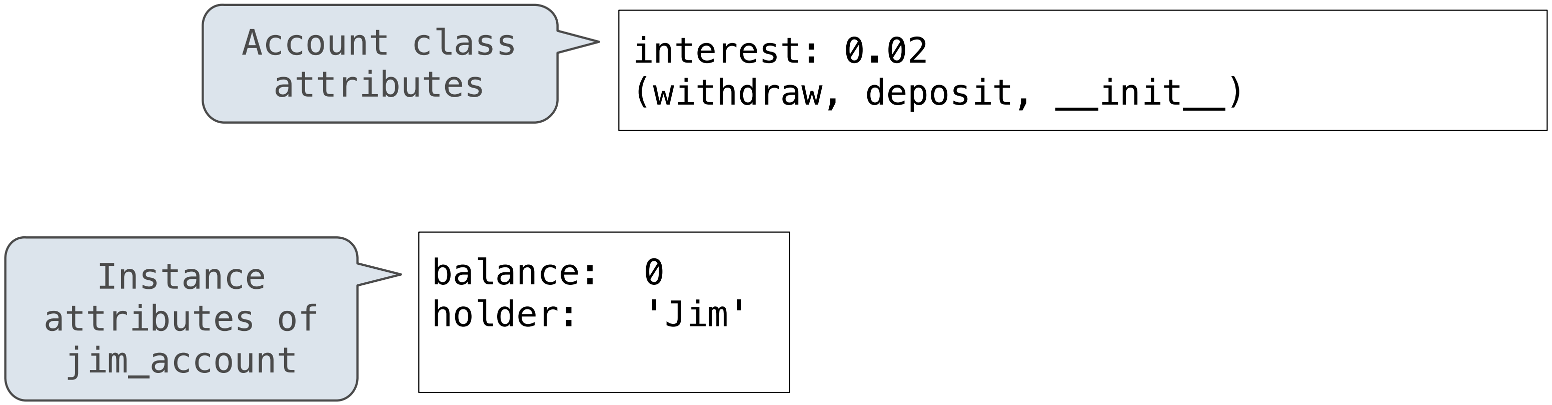
```
interest: 0.02
(withdraw, deposit, __init__)
```

Instance attributes of jim_account

```
balance:   0
holder:    'Jim'
```

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

Attribute Assignment Statements



Account class attributes

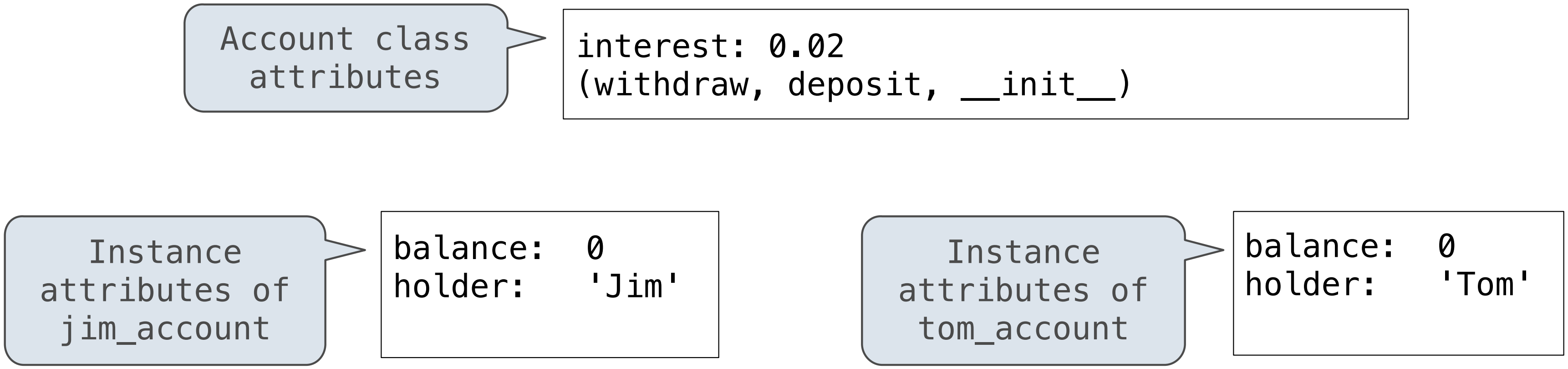
```
interest: 0.02
(withdraw, deposit, __init__)
```

Instance attributes of jim_account

```
balance:   0
holder:    'Jim'
```

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

Attribute Assignment Statements



Account class attributes

```
interest: 0.02
(withdraw, deposit, __init__)
```

Instance attributes of jim_account

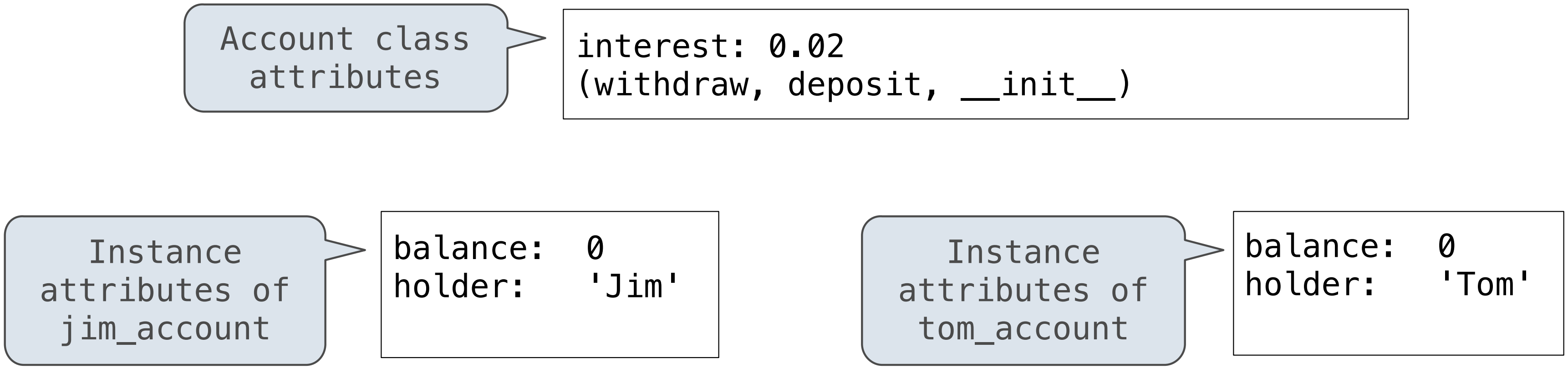
```
balance:  0
holder:   'Jim'
```

Instance attributes of tom_account

```
balance:  0
holder:   'Tom'
```

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

Attribute Assignment Statements



Account class attributes

```
interest: 0.02
(withdraw, deposit, __init__)
```

Instance attributes of jim_account

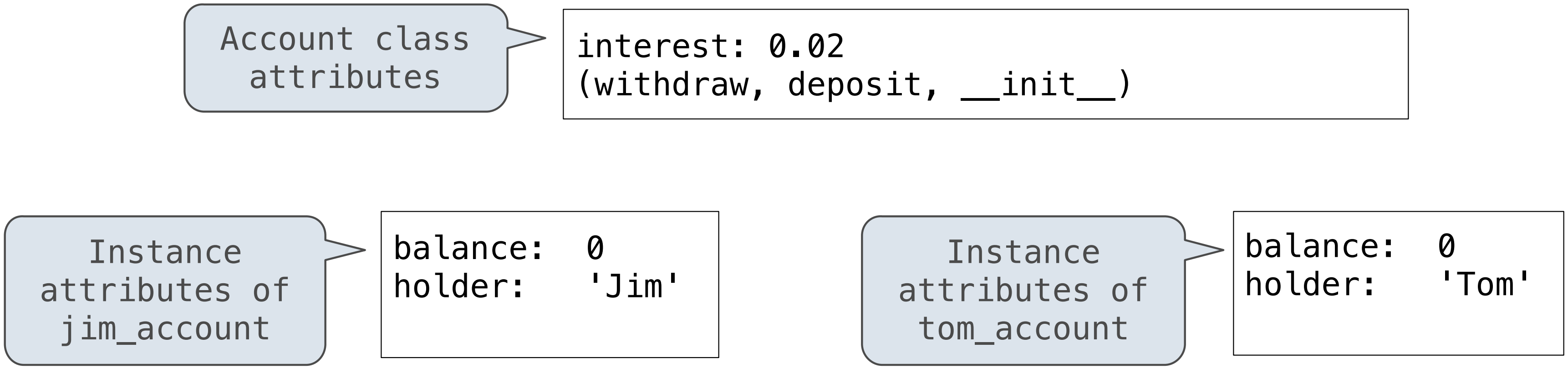
```
balance:  0
holder:   'Jim'
```

Instance attributes of tom_account

```
balance:  0
holder:   'Tom'
```

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

Attribute Assignment Statements



Account class attributes

```
interest: 0.02
(withdraw, deposit, __init__)
```

Instance attributes of jim_account

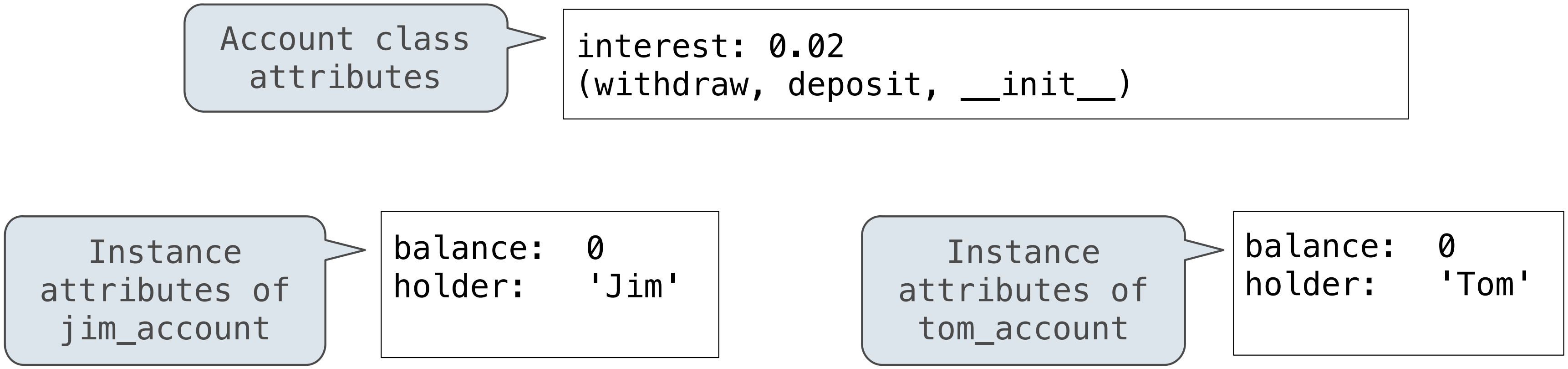
```
balance:   0
holder:    'Jim'
```

Instance attributes of tom_account

```
balance:   0
holder:    'Tom'
```

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

Attribute Assignment Statements



Account class attributes

```
interest: 0.02
(withdraw, deposit, __init__)
```

Instance attributes of jim_account

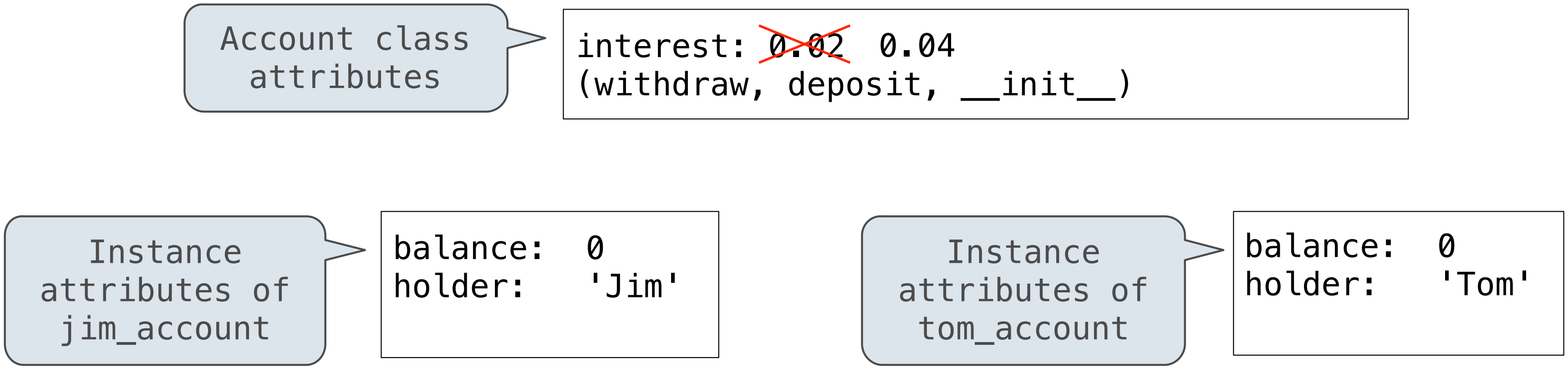
```
balance:  0
holder:   'Jim'
```

Instance attributes of tom_account

```
balance:  0
holder:   'Tom'
```

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

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~ 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

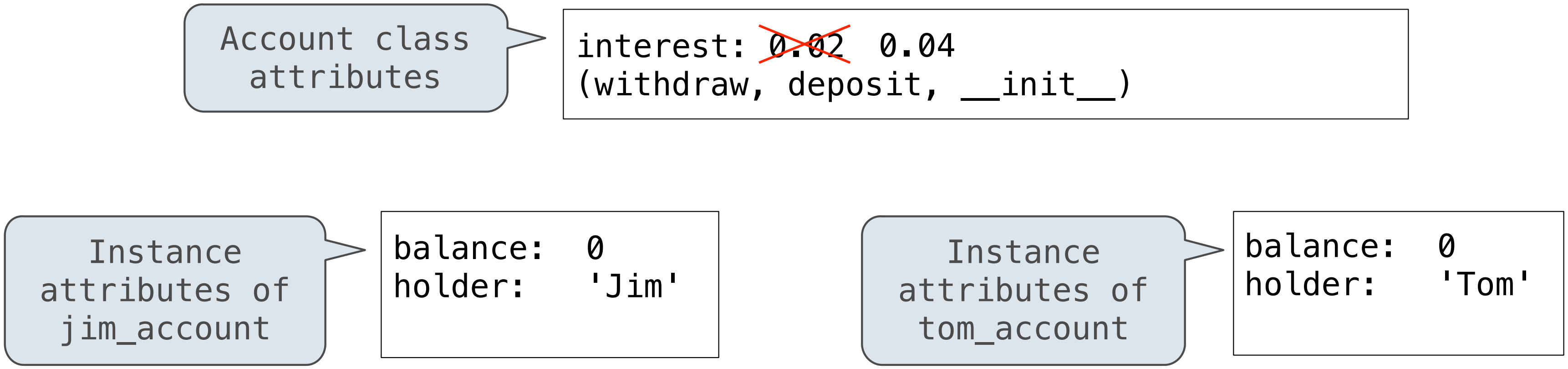
balance: 0
holder: 'Jim'

Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~ 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

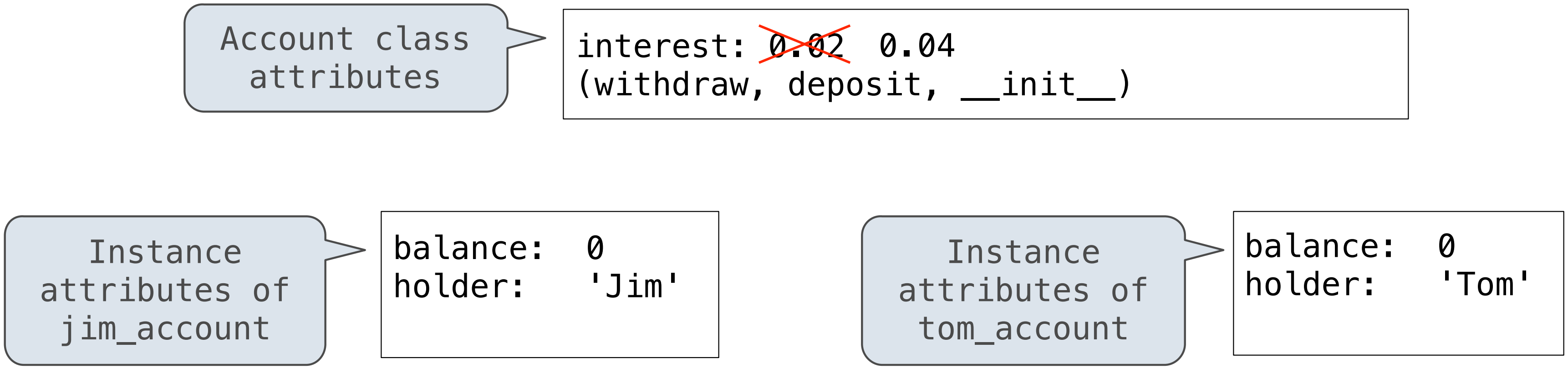
balance: 0
holder: 'Jim'

Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~ 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

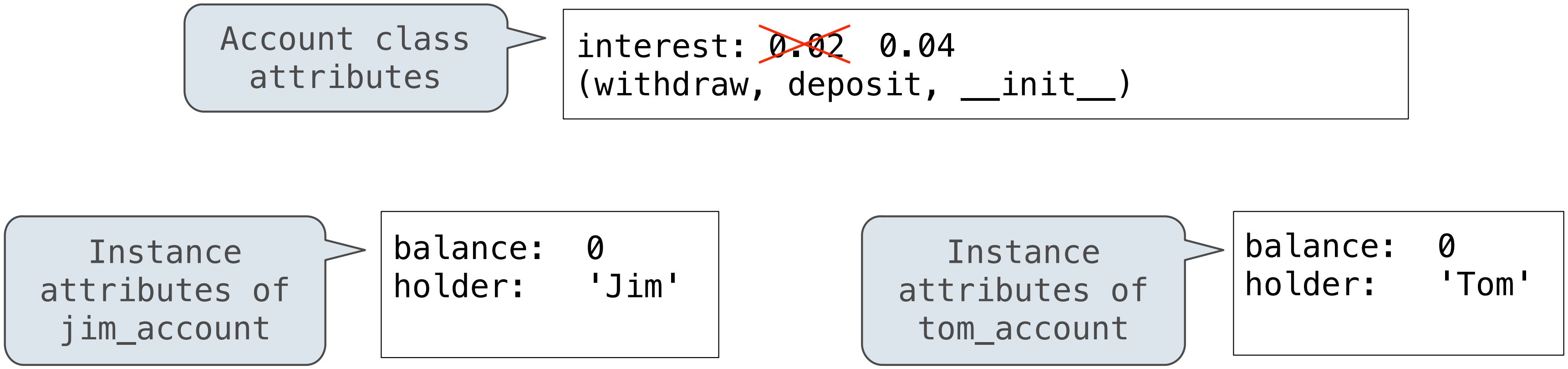
balance: 0
holder: 'Jim'

Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~ 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'

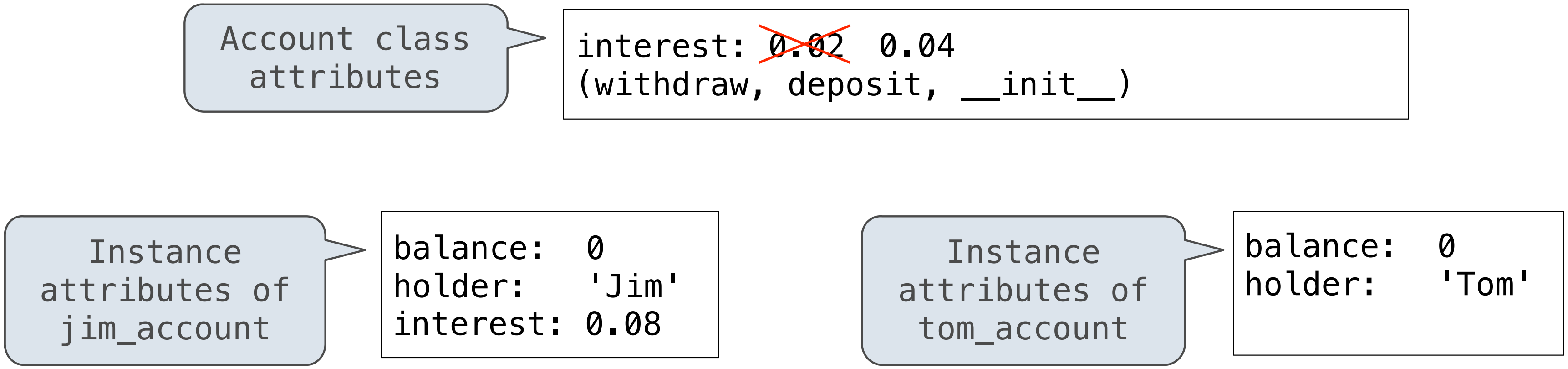
Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

```
>>> jim_account.interest = 0.08
```

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~ 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

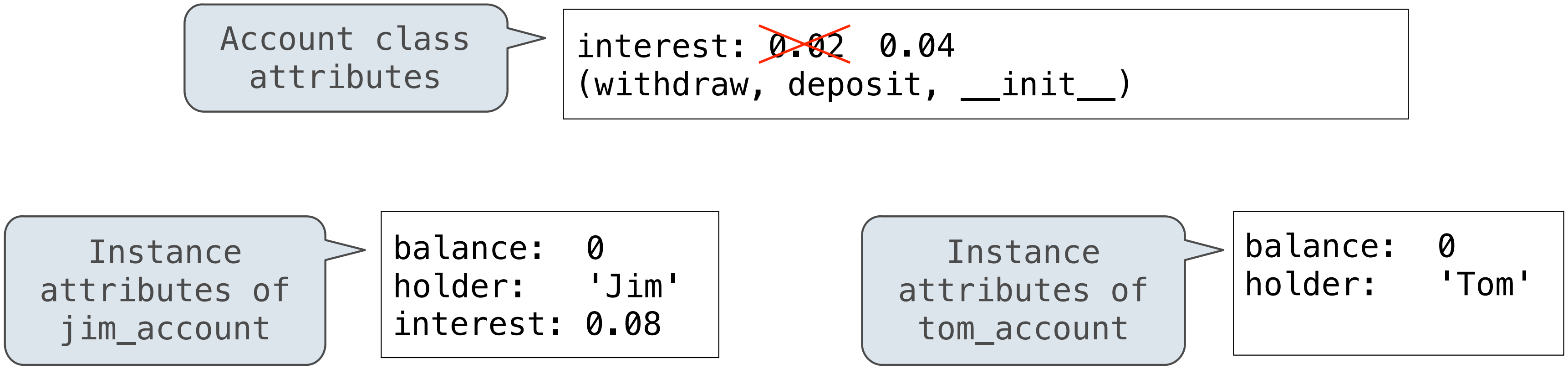
Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

```
>>> jim_account.interest = 0.08
```

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~ 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

```
balance:   0
holder:    'Jim'
interest: 0.08
```

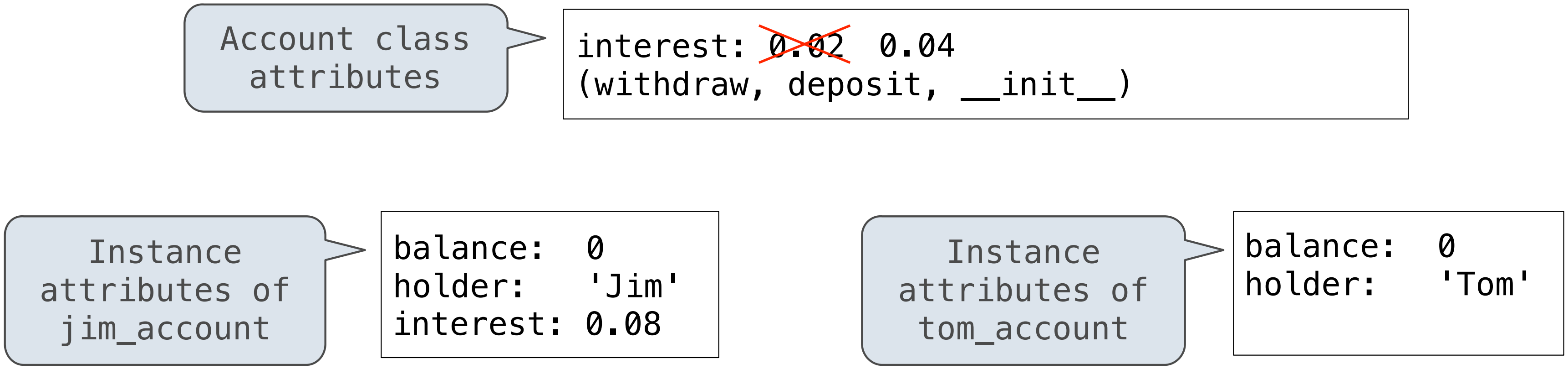
Instance attributes of tom_account

```
balance:   0
holder:    'Tom'
```

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

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
```

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~ 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

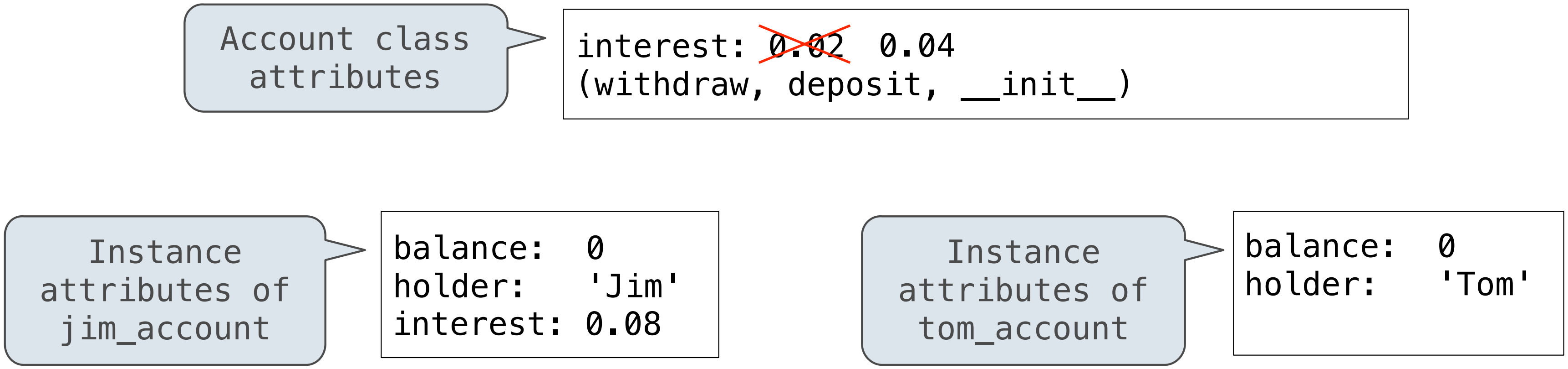
Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
```

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~ 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

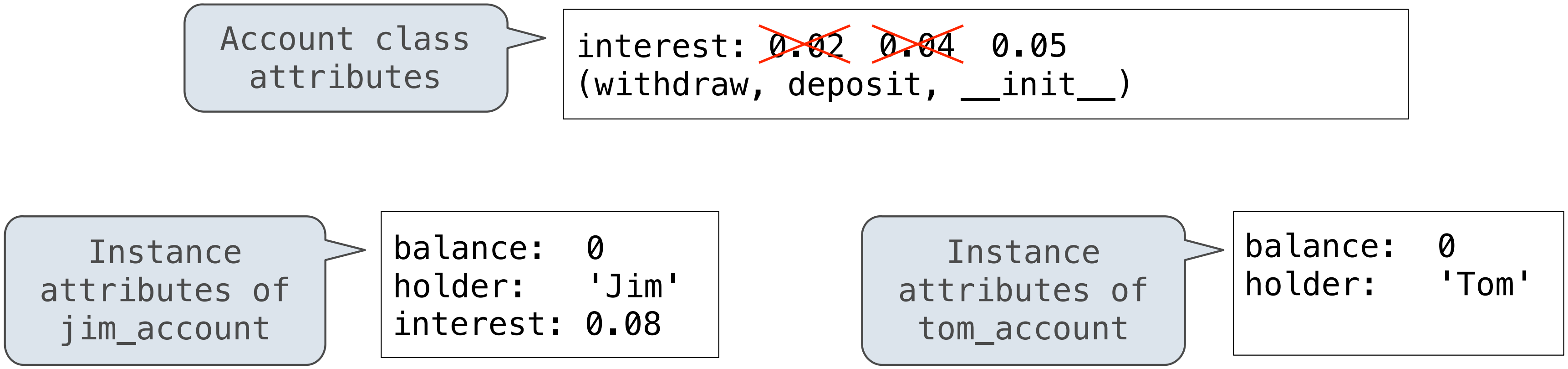
Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
```

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~ ~~0.04~~ 0.05
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

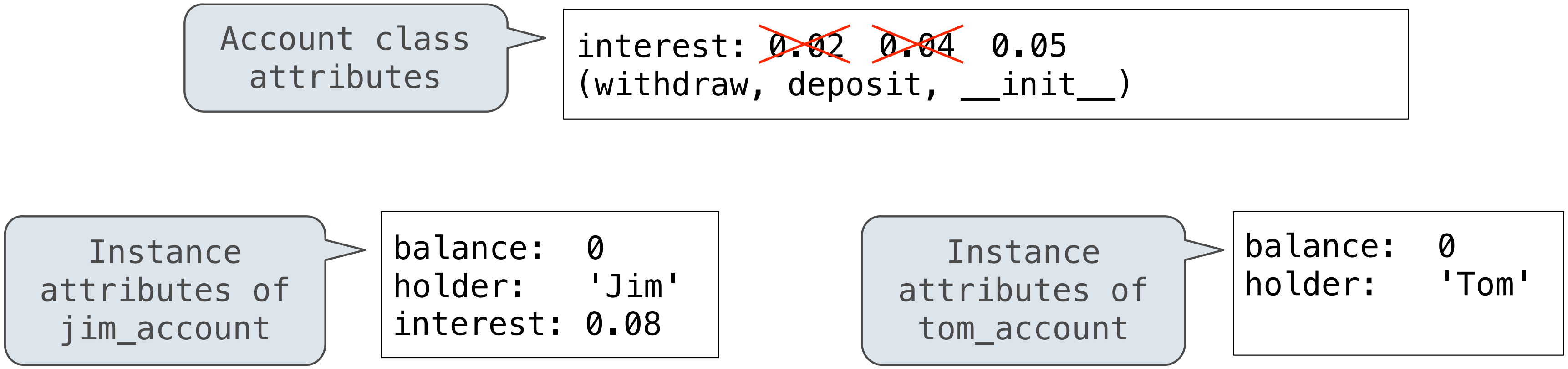
Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
```

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~ ~~0.04~~ 0.05
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

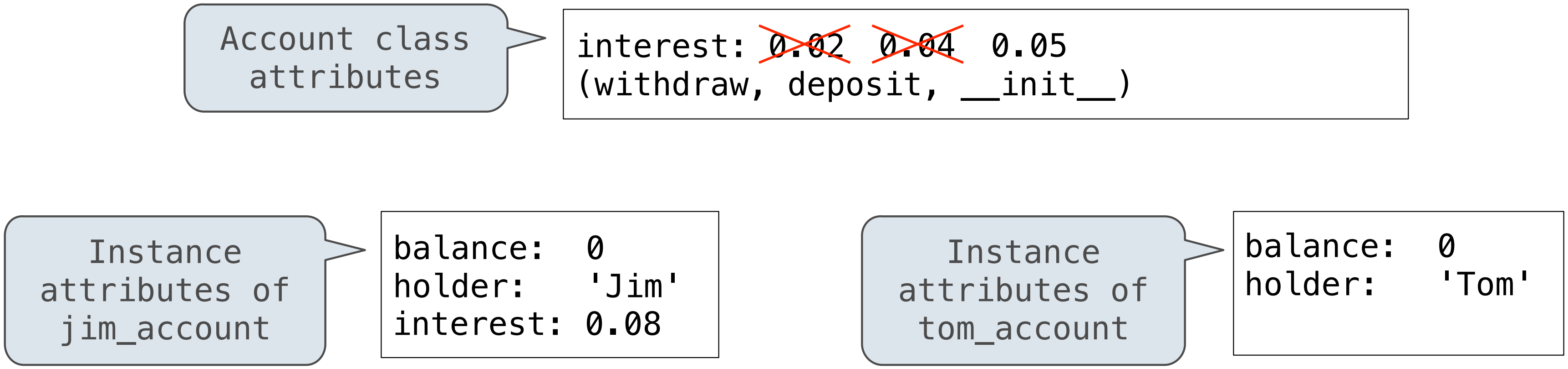
Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
```

Attribute Assignment Statements



Account class attributes

```
interest: ~~0.02~~ ~~0.04~~ 0.05
(withdraw, deposit, __init__)
```

Instance attributes of jim_account

```
balance:   0
holder:    'Jim'
interest: 0.08
```

Instance attributes of tom_account

```
balance:   0
holder:    'Tom'
```

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

```
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.08
```