Object Oriented Programming

Lecture 12 -13

Before: Procedural programming

- The entire program was divided into smaller parts
  - A.k.a. Functions
- Data was not important, the sequence of actions was more important
- Usually the process was top-bottom.

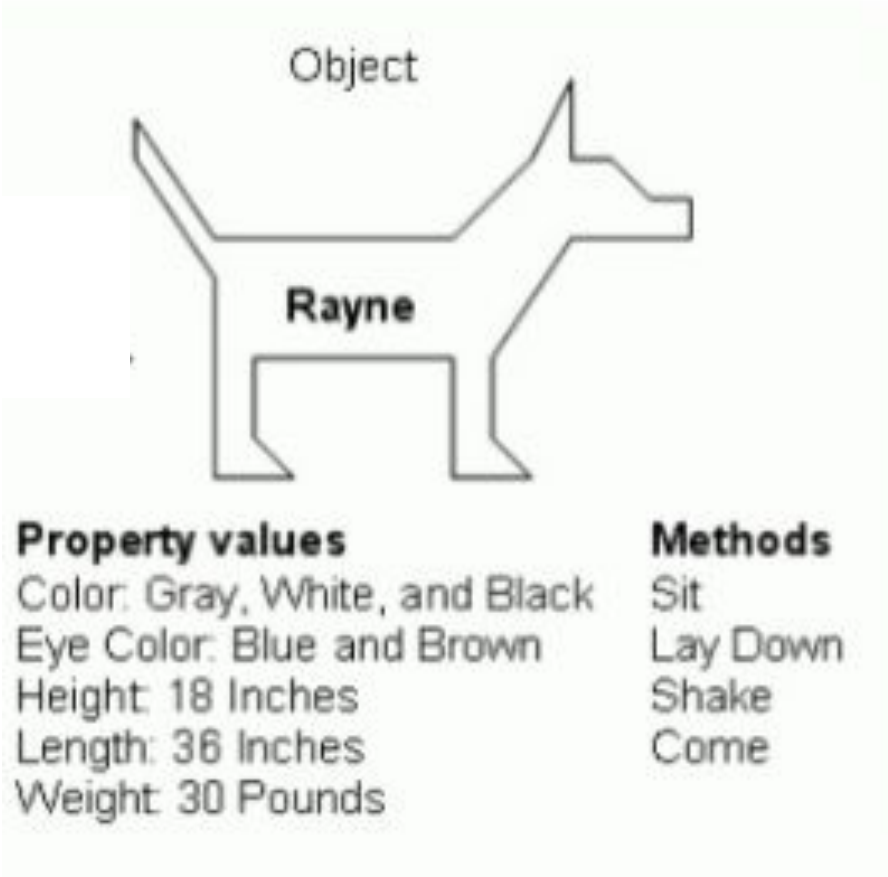
Action before Data

Now: Object Oriented Programming

Data (types) before Action

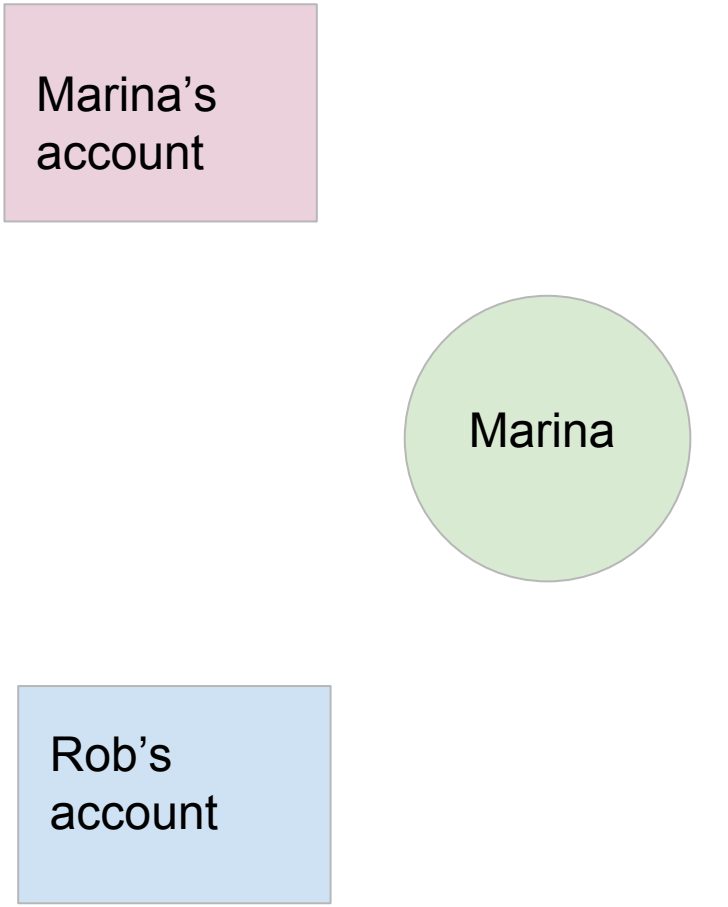
Objects

(demo)

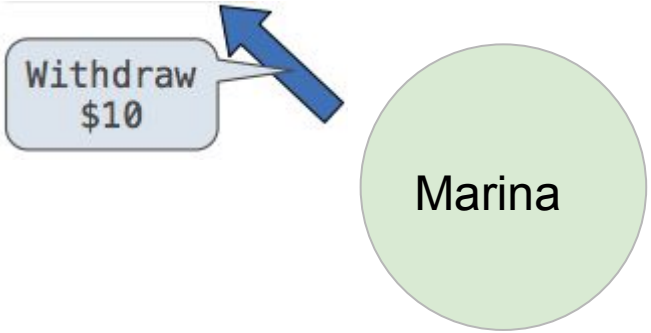

Object
Rayne
Property values
Color: Gray, White, and Black
Eye Color: Blue and Brown
Height: 18 Inches
Length: 36 Inches
Weight: 30 Pounds
Methods
Sit
Lay Down
Shake
Come

Object-Oriented Programming

- **A method for organizing modular 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 and vocabulary to support this metaphor**

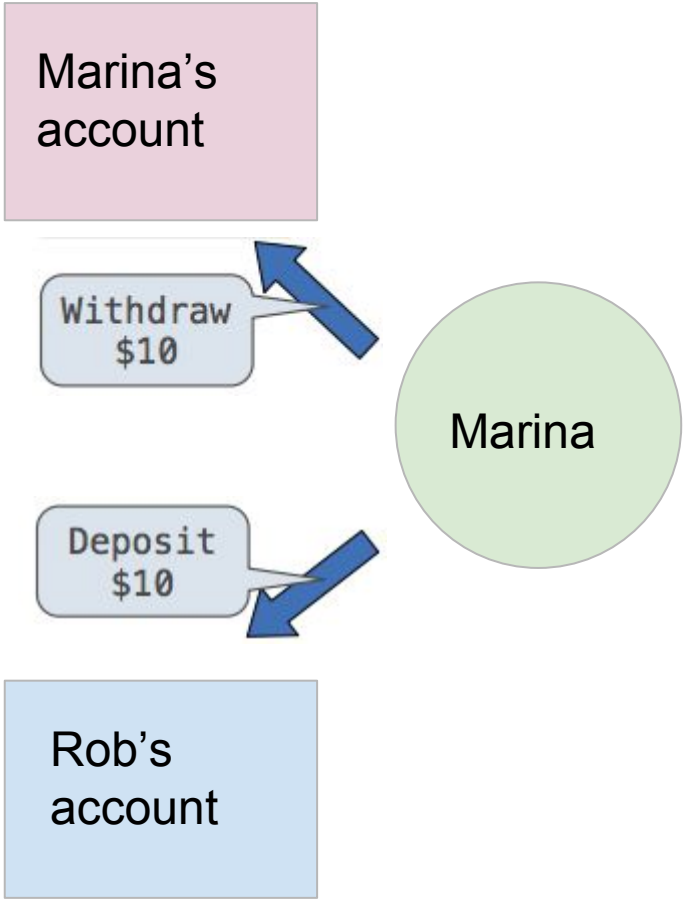

Marina’s account
Marina
Rob’s account

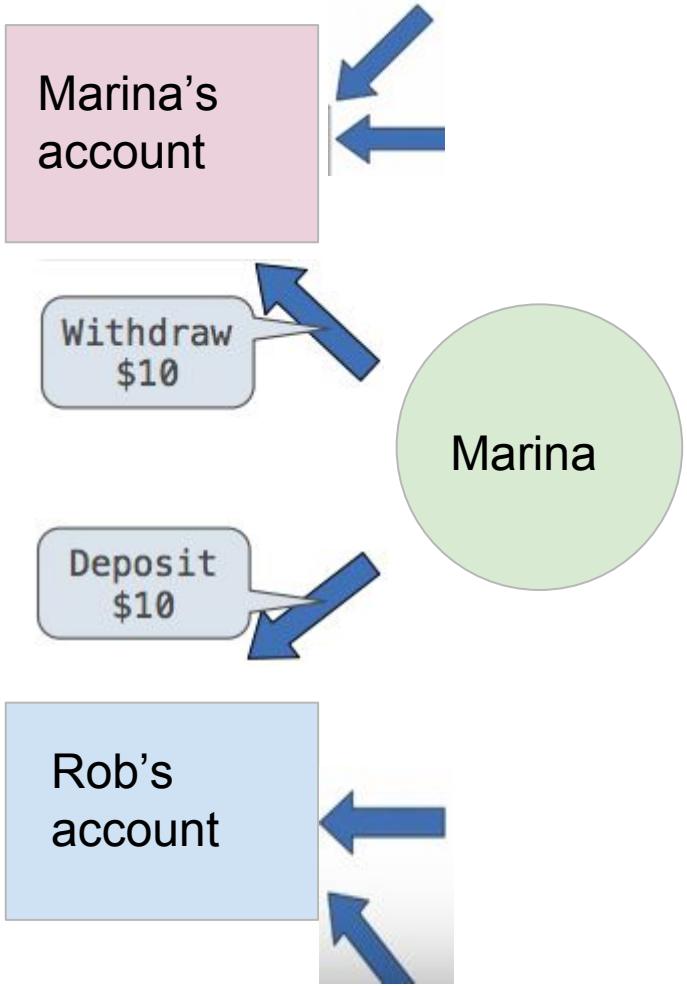
Marina's account

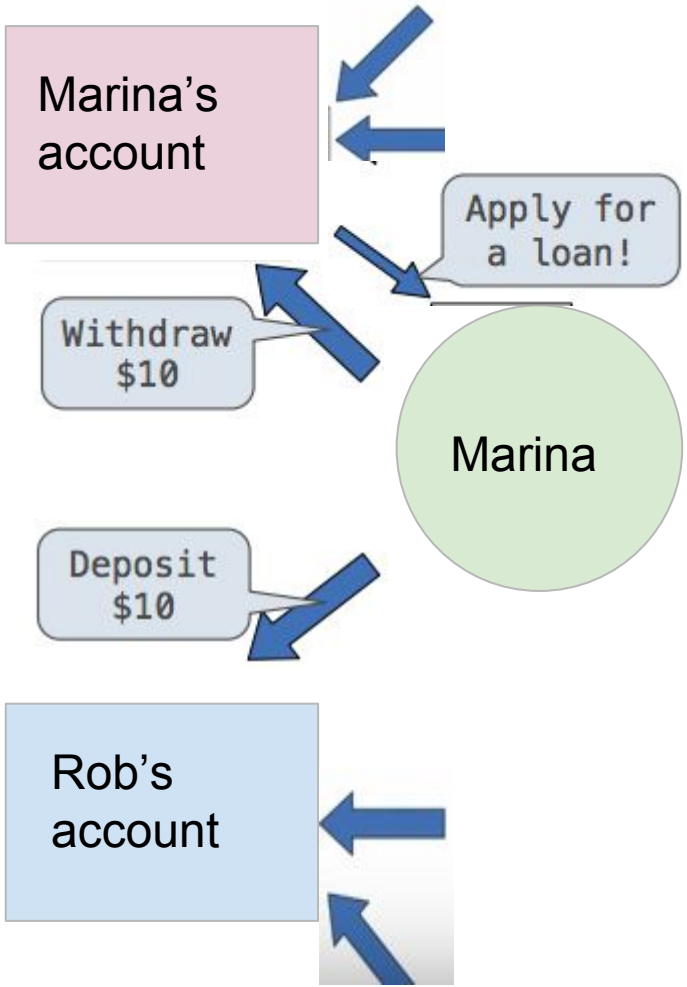


Message passed from Marina to Account

Rob's account

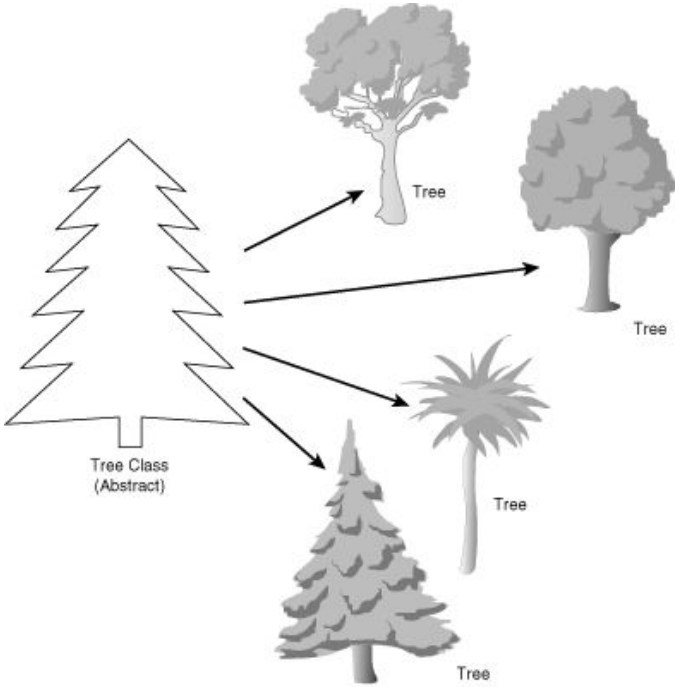

Marina’s account
Withdraw $10
Marina
Deposit $10
Rob’s account


Marina’s account
Withdraw $10
Marina
Deposit $10
Rob’s account


Marina's account
Apply for a loan!
Withdraw $10
Marina
Deposit $10
Rob's account

Classes

- A class serves as a **template** for its **instances**
- Each object is an *instance* of some class



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('Marina')
```

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('Marina')
>>> a.holder      #attribute of that particular account
'Marina'
```

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('Marina')
>>> a.holder      #attribute
'Marina'
>>> a.balance
0
```

Classes

- A class serves as a template for its instances

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

```
>>> a = Account('Marina')
>>> a.deposit(15)
```

Classes

- A class serves as a template for its instances

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

```
>>> a = Account('Marina')
>>> a.deposit(15)
15
```

Classes

- A class serves as a template for its instances

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

```
>>> a = Account('Marina')
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
```

Classes

- A class serves as a template for its instances

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

```
>>> a = Account('Marina')
>>> 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 should have `withdraw` and `deposit` behaviors that all work in the same way.

```
>>> a = Account('Marina')
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.balance
5
```

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

The Class Statement: any type of data

```
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* and *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* and *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 and def statements in **<suite>** create attributes of the class (not names in frames)

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

Attributes of the class

```
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 and 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'
```

Attributes of the class

```
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 and 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('Rob')
```

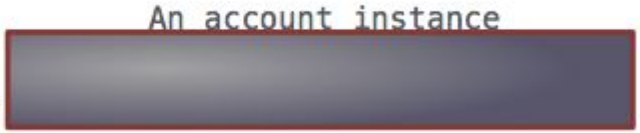
Object Construction

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('Rob')
```

When a class is called:

1. A new instance of that class is created:



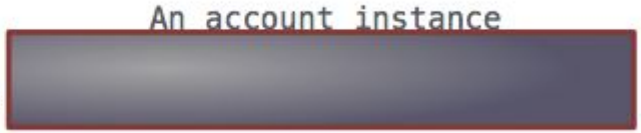
Object Construction

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('Rob')
```

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

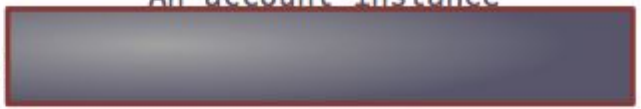
Object Construction

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('Rob')
```

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
```

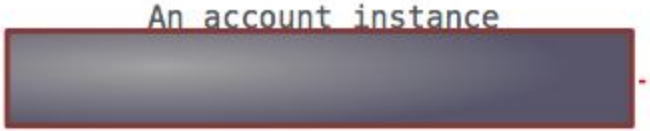
Object Construction

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('Rob')
```

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

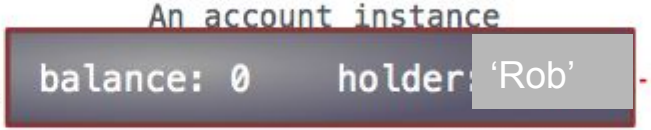
```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```

Object Construction

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

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

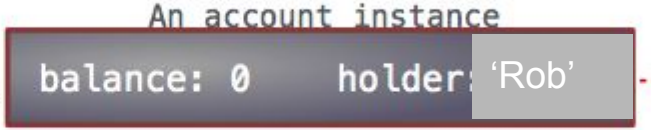
```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```

Object Construction

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

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')
```

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.

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:

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

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
```

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

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 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

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

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

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

```
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
```

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

Check point

Which statement is true?

A: A class is blueprint for the object.

B: You can only make a single object from the given class.

C: Both statements are true.

D: Neither statement is true.

Check point

What does the `__init__()` function do in Python?

A: Initializes the class for use.

B: It is called when a new object is created.

C: Initializes all the attributes to zero when called.

D: INone of the above.

Check point

```
class Point:
    def __init__(self, x, y):
        self.x = x+1
        y = y+1

p1 = Point(0,  0)
print(p1.x, p1.y)
```

A: 0, 0

B: 1, 1

C: Not enough information

D: Error

Attributes
(data that stored within either an instance or the class itself)

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

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

Example

```
class Example:

    count = 1
    another_numer = 6

    def __init__(self, initial_value):
        self.count = 0
        self.count = self.count + initial_value

    def increment(self, amount):
        self.count = self.count + amount
        count = count + 1
        return self.count
```

```
e = Example(4)
print(e.count)
```

Example

```
class Example:

     count = 1
     another_numer = 6

     def __init__(self, initial_value):
          self.count = 0
          self.count = self.count + initial_value

     def increment(self, amount):
          self.count = self.count + amount
          count = count + 1
          return self.count
```

```
e = Example(4)
print(e.count)
print(Example.count)
```

Example

```
class Example:

     count = 1
     another_numer = 6

     def __init__(self, initial_value):
          self.count = 0
          self.count = self.count + initial_value

     def increment(self, amount):
          self.count = self.count + amount
          Example.count = Example.count + 1
          return self.count
```

```
e = Example(4)
print(e.count)
print(Example.count)
print(e.another_numer)
e.increment(10)
print (e.count)
print (Example.count)
```

Methods and Functions

Python distinguishes between:

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

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'>
```

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

Methods and Functions

Object + Function = Bound Method

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

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

```
>>> Account.deposit(tom_account, 1001)
1011
```

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

Methods and Functions

Object + Function = Bound Method

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

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

```
>>> Account.deposit(tom_account, 1001)
1011
>>> tom_account.deposit(1004)
2015
```

Methods and Functions

Object + Function = Bound Method

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

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

```
>>> Account.deposit(tom_account, 1001)
1011
>>> tom_account.deposit(1004)
2015
```

Function: all arguments within parentheses

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

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

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!

Question

```
class Clown:

    nose = 'big and red'

    def __init__(self, clown_name):
        self.name = clown_name
        self.salary = 3000

    def dance():
        return 'No thanks'


c = Clown("Marina")
Clown.nose = 'blue and small'
t = Clown("Tom")
Clown.nose = "blue and pink"
```

	t.nose	c.nose
A:	blue, pink	blue, pink
B:	blue, small	blue, small
C:	blue, pink	blue, small
D:	blue, small	blue, pink
E:	Something else	

Question

```
class Clown:

    nose = 'big and red'
    def __init__(self, clown_name):
        self.name = clown_name
        self.salary = 3000
    def dance():
        return 'No thanks'

c=Clown("Marina")
c.nose = 'blue and small'
t = Clown("Tom")
Clown.nose = "blue and pink"
t.nose
```

	t.nose	c.nose
A:	blue, pink	blue, pink
B:	blue, small	blue, small
C:	blue, pink	blue, small
D:	blue, small	blue, pink
E:	Something else	

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

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
```

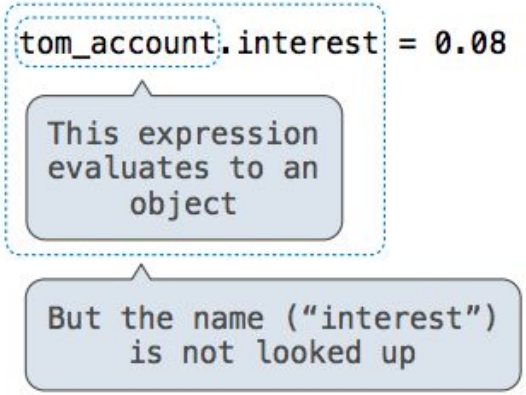
This expression evaluates to an object

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')
```

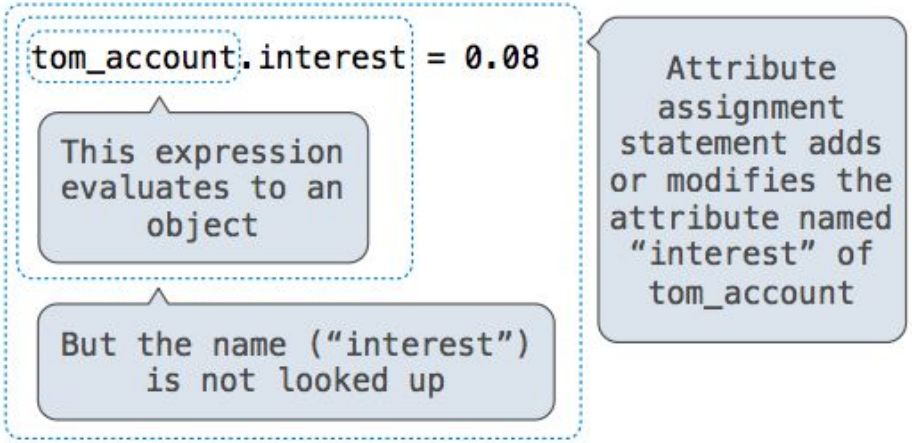


Assignment to Attributes

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

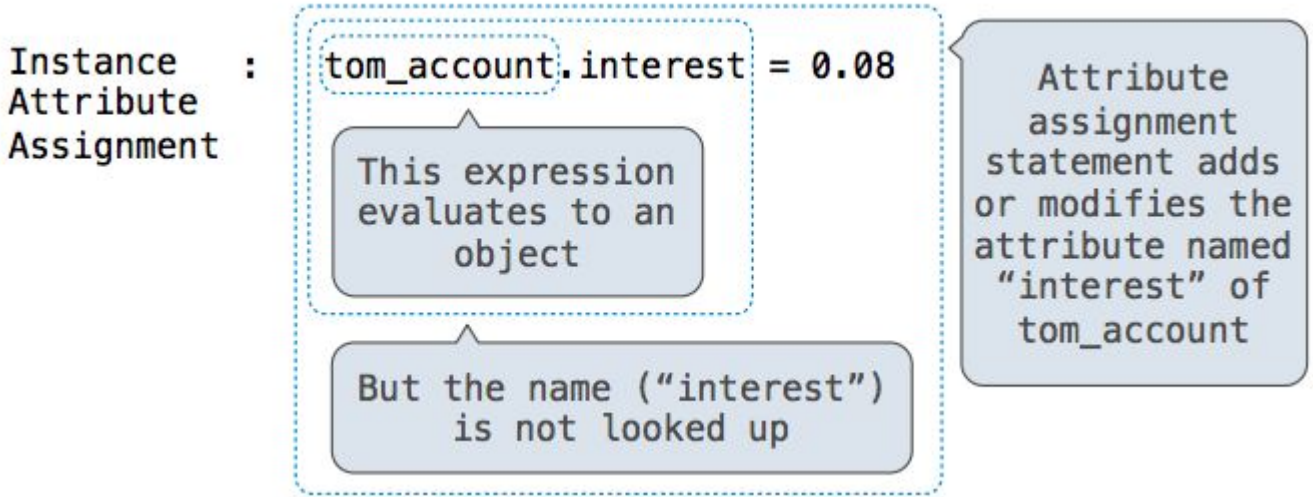
- If the object is an instance, then assignment sets an instance attribute
- If the object is a class, then assignment sets a class attribute

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

tom_account = Account('Tom')
```



Assignment to Attributes

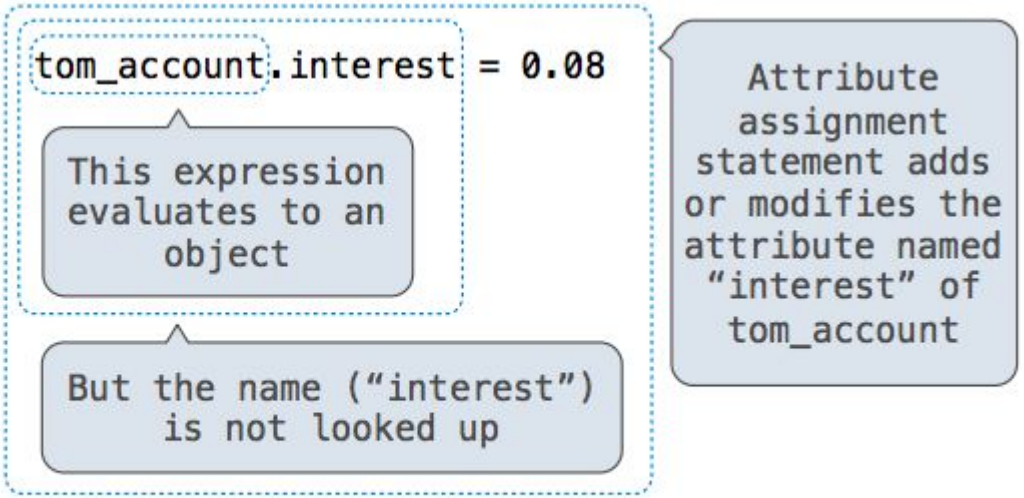
```
class Account:
    interest = 0.02
    def __init__(self, holder):
        self.holder = holder
        self.balance = 0
    ...

tom_account = Account('Tom')
```



Assignment to Attributes

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

tom_account = Account('Tom')
```

Instance Attribute Assignment :



Class Attribute Assignment : Account.interest = 0.04

Account class attributes

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

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

Account class attributes

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

Instance attributes of jim_account

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

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

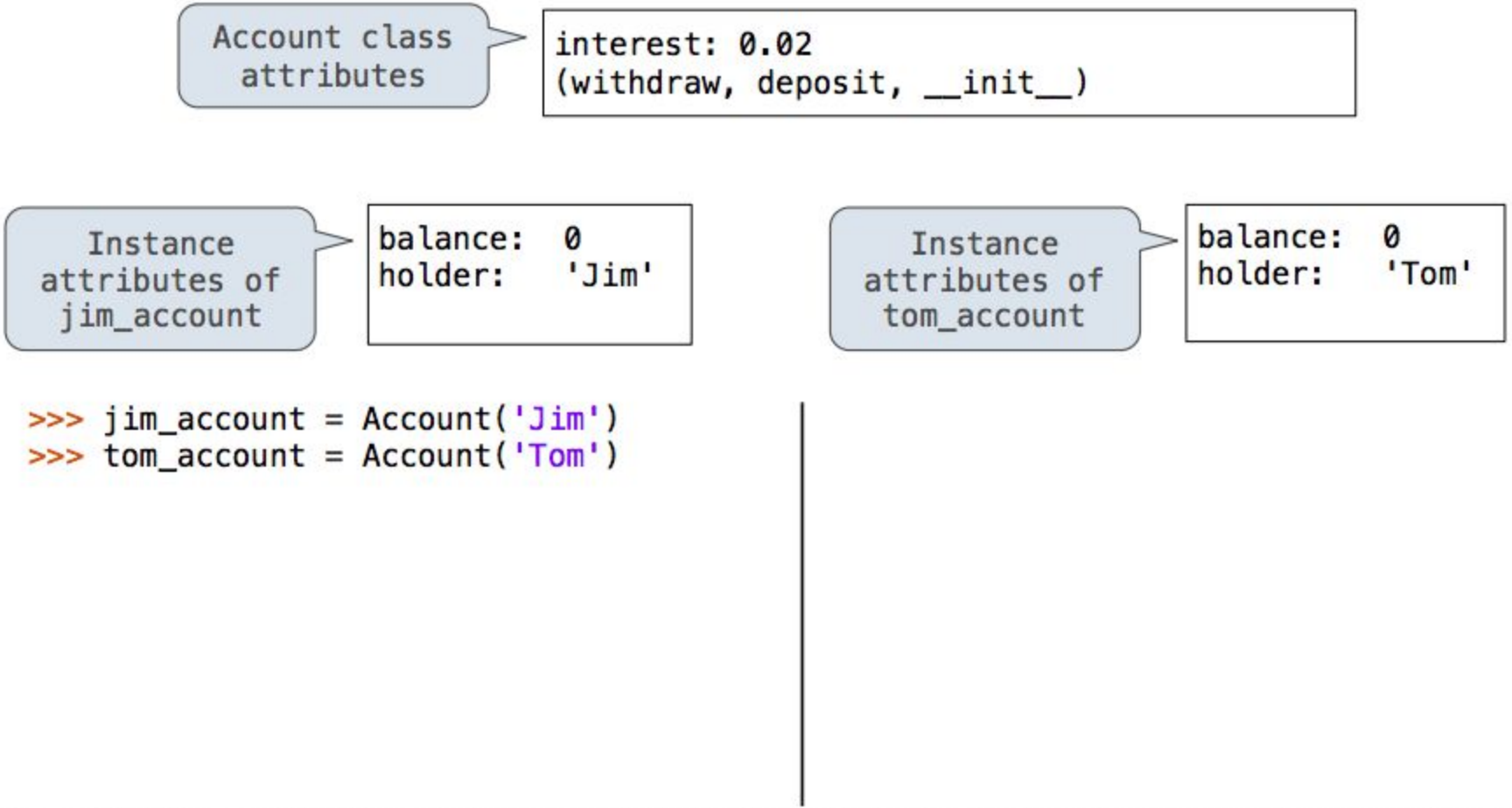
Account class attributes

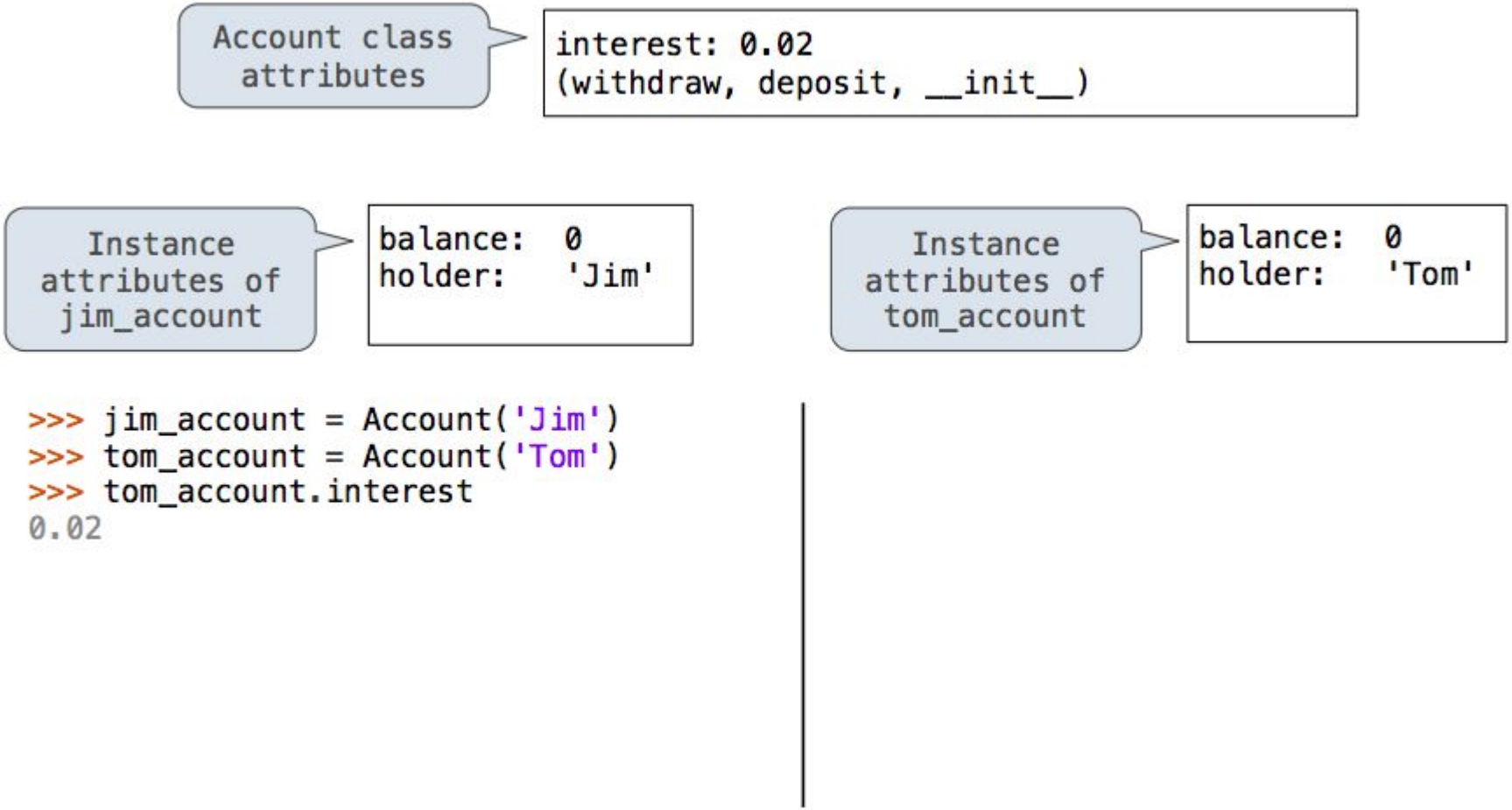
```
interest: 0.02
(withdraw, deposit, __init__)
```

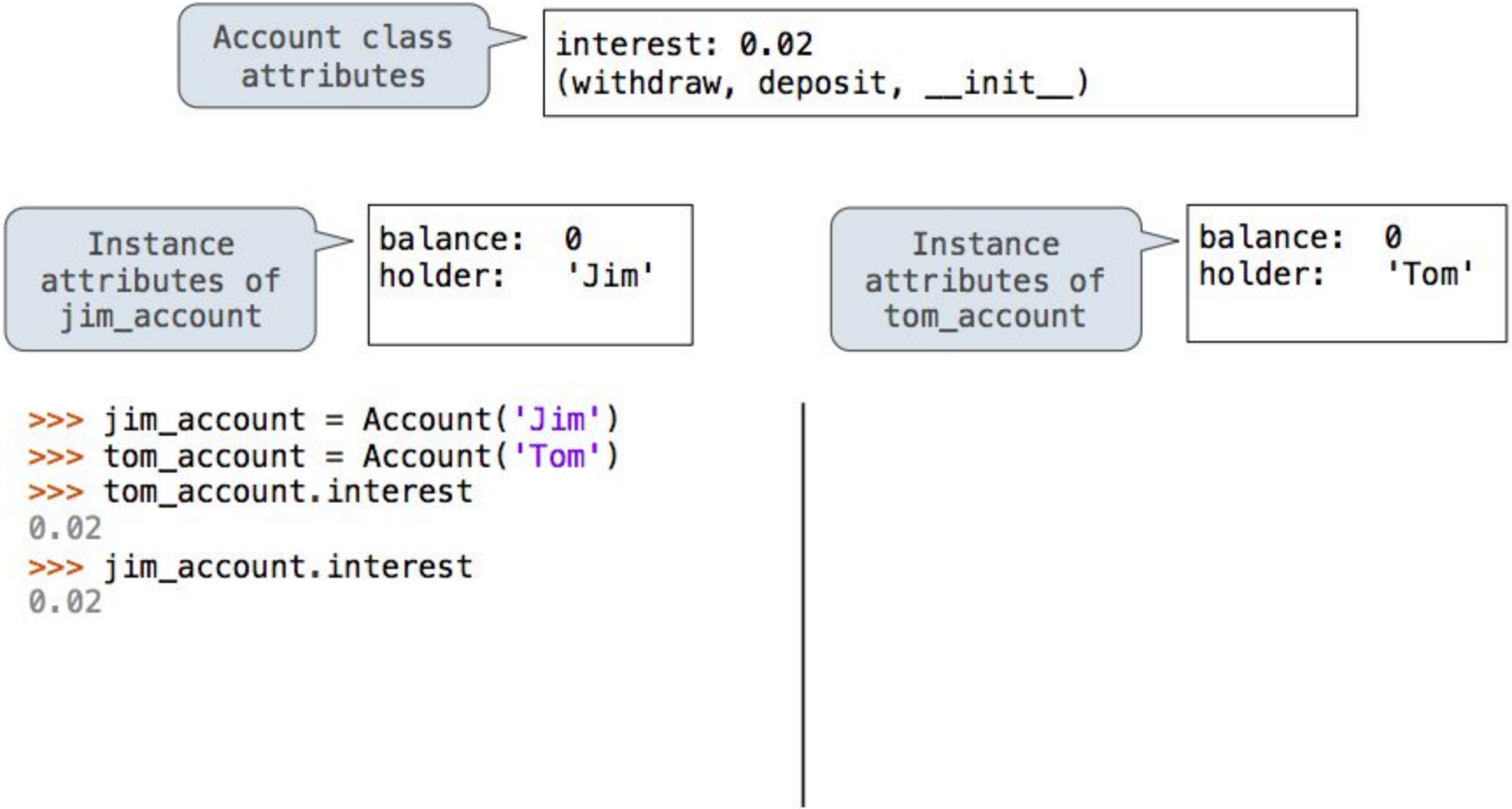
Instance attributes of jim_account

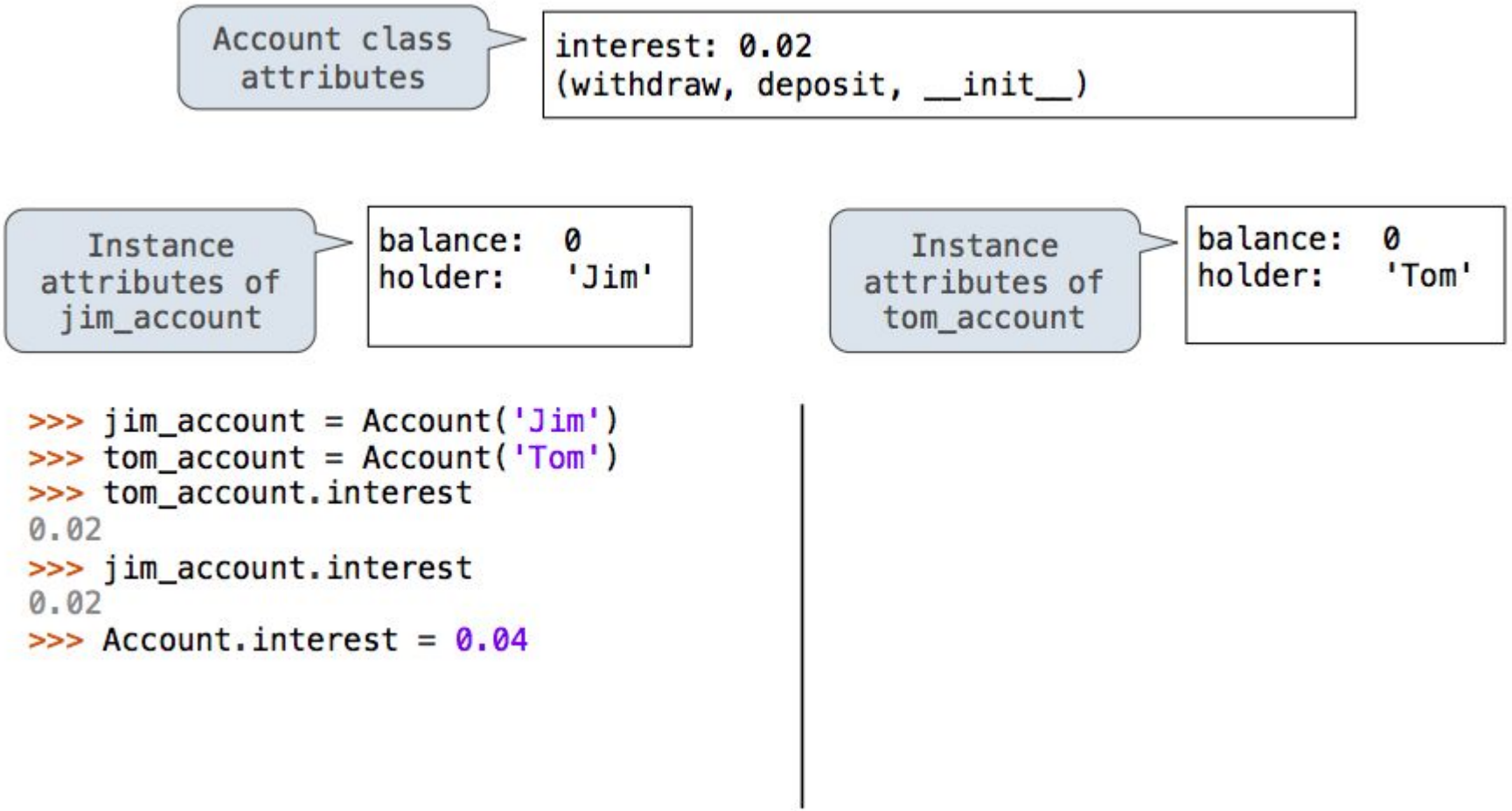
```
balance:  0
holder:   'Jim'
```

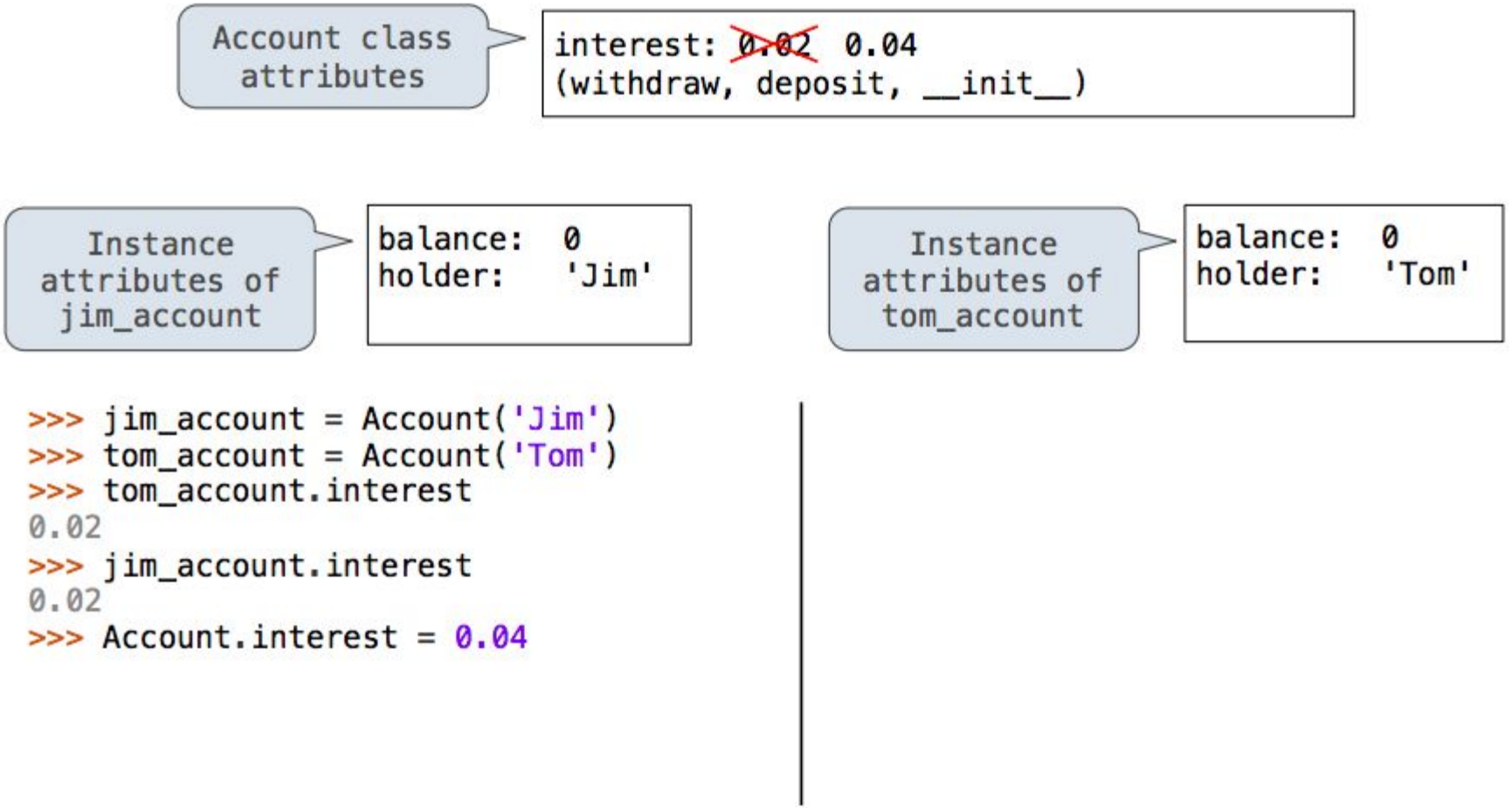
```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
```


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')


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


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 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


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

Account class attributes

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

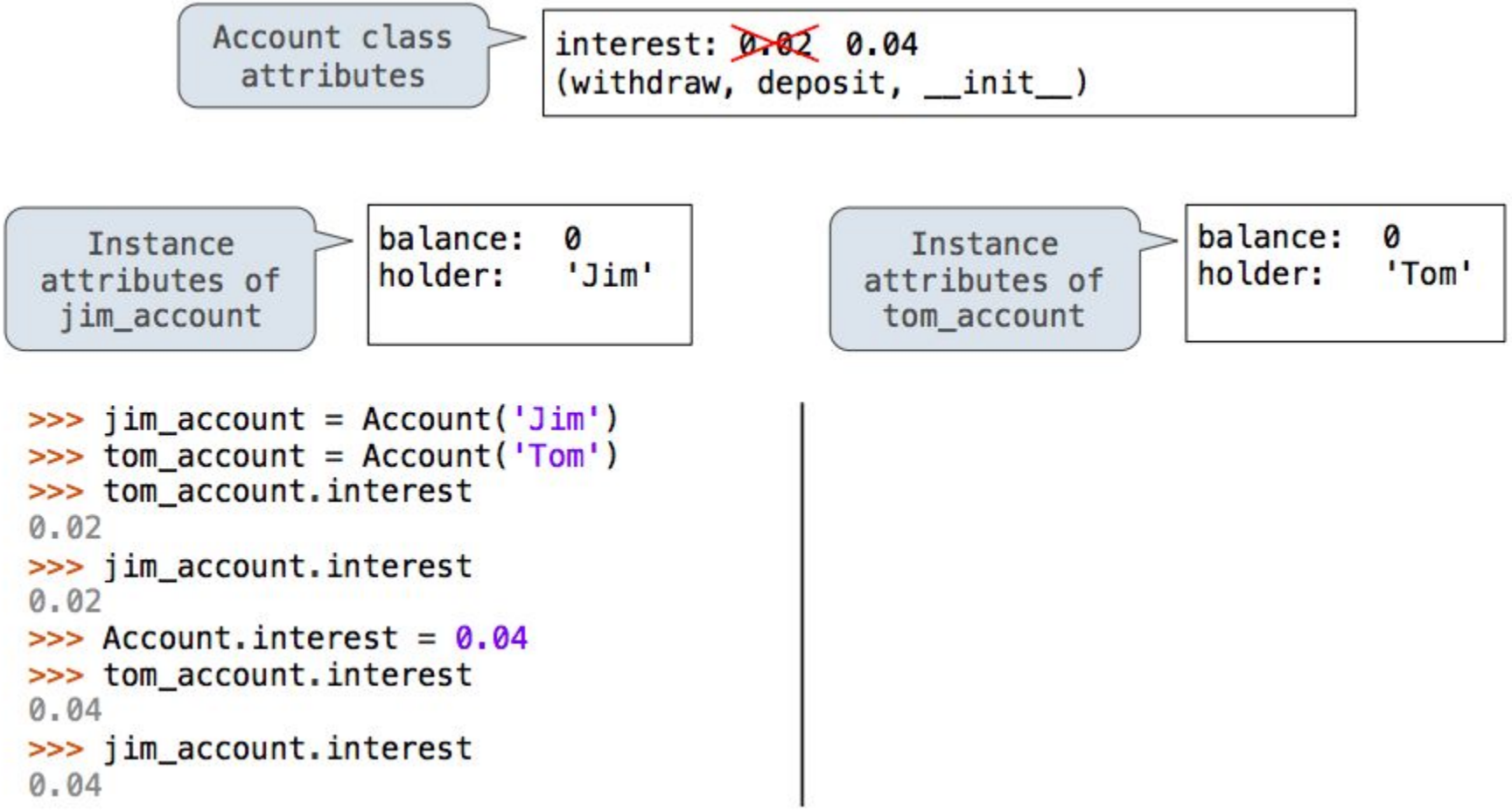
Instance attributes of jim_account

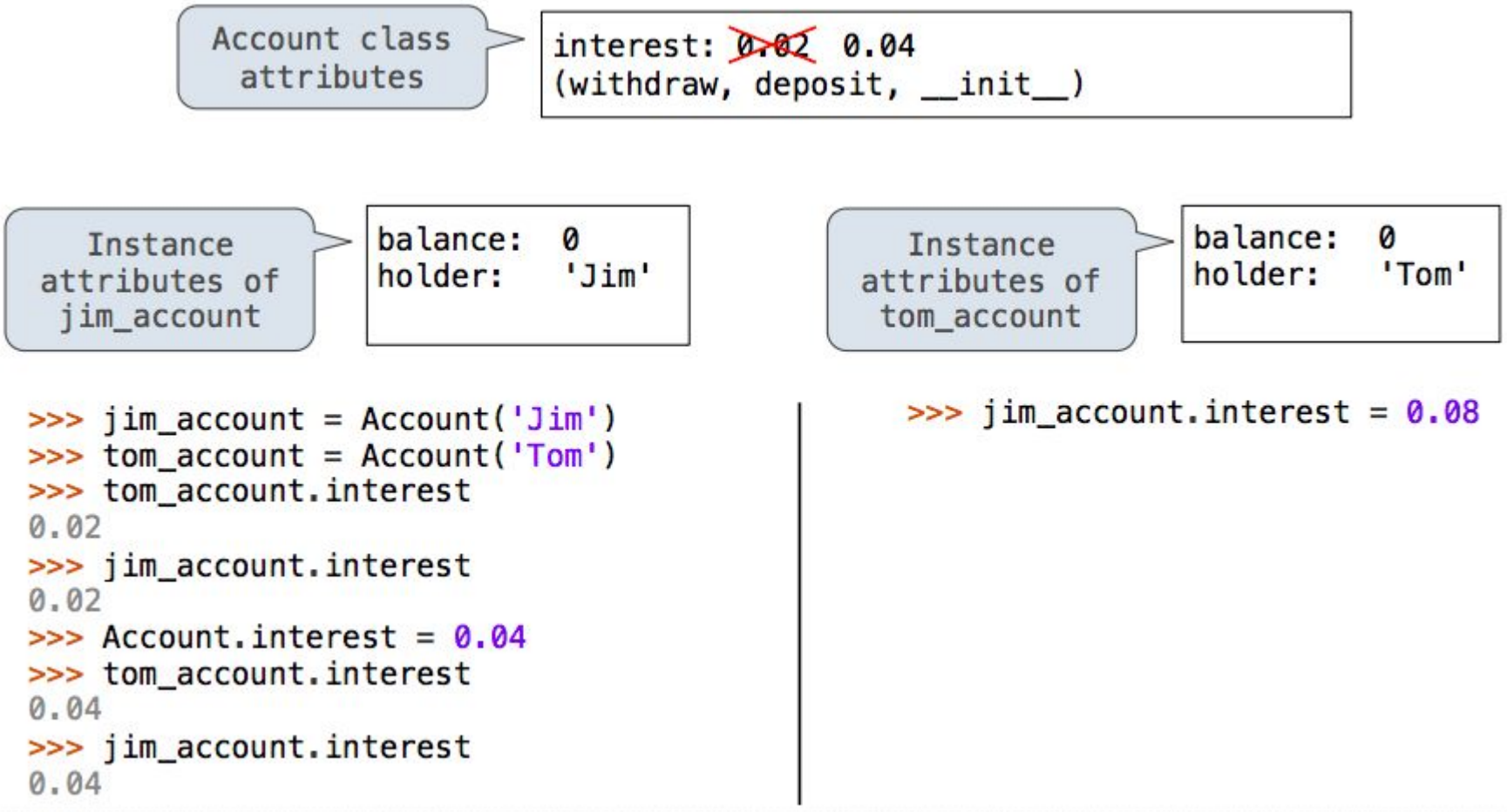
balance: 0
holder: 'Jim'

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
```


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


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

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
```

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
```

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 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
```

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
```

Account class attributes

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

Instance attributes of jim_account

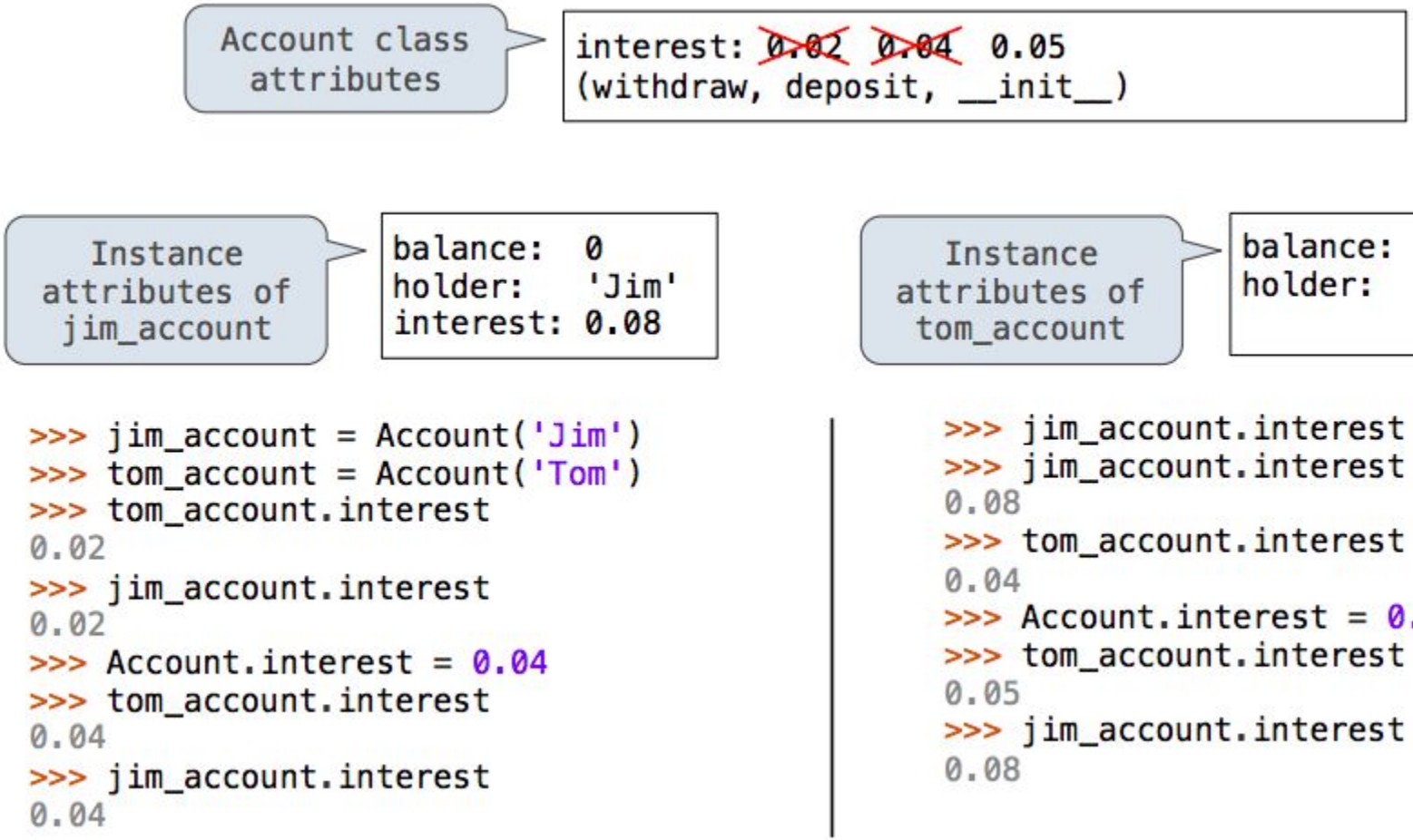
```
balance:  0
holder:   'Jim'
interest: 0.08
```

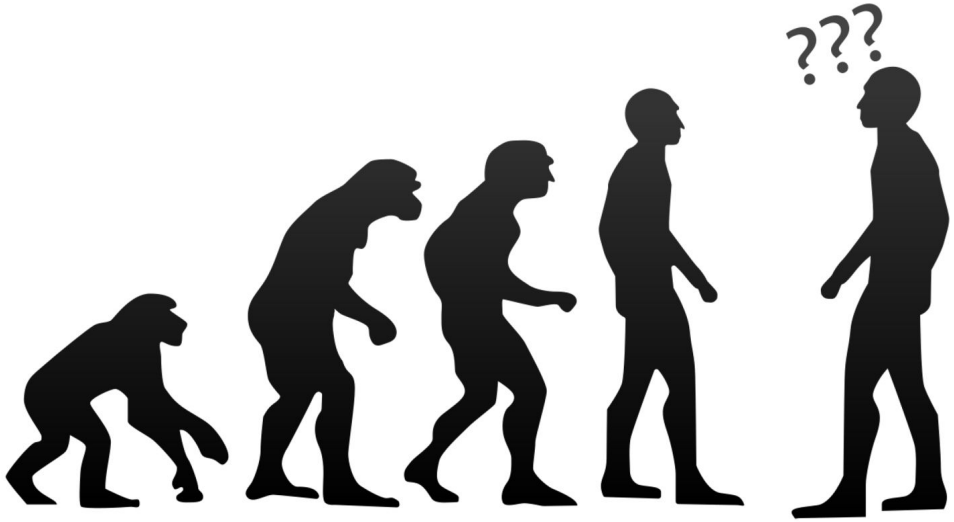
Instance attributes of tom_account

```
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
```


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



Inheritance

Inheritance

- Inheritance is a technique for relating classes together
- **A common use:** Two similar classes differ in their degree of specialization
- The specialized class may have the same attributes as the general class, along with some special-case behavior

```
class <Name>(<Base Class>):
    <suite>
```

Inheritance

```
class <Name>(<Base Class>):
    <suite>
```

- Conceptually, the new *subclass* (child class) inherits (shares) attributes of its base (parent, super) class
- The subclass may *override* certain inherited attributes
- Using inheritance, we implement a subclass by specifying its differences from the the base class

Inheritance Example

A CheckingAccount is a *specialized* type of Account

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
```

Inheritance Example

A CheckingAccount is a specialized type of Account

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
```

Inheritance Example

A CheckingAccount is a specialized type of Account

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```

Most behavior is shared with the base class Account

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
```

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
```

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
```

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
```

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
```

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
                                or
        return super().withdraw(      amount + self.withdraw_fee)
```

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
```

or

```
        return super().withdraw(      amount + self.withdraw_fee)
```

Looking Up Attribute Names on Classes

Base class attributes aren't copied into subclasses!

To look up a name in a class:

1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

```
>>> ch = CheckingAccount('Tom')  # Calls Account.__init__
>>> ch.interest     # Found in CheckingAccount
0.01
```

Looking Up Attribute Names on Classes

Base class attributes aren't copied into subclasses!

To look up a name in a class:

1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

```
>>> ch = CheckingAccount('Tom')  # Calls Account.__init__
>>> ch.interest      # Found in CheckingAccount
0.01
>>> ch.deposit(20)  # Found in Account
20
```

Looking Up Attribute Names on Classes

Base class attributes aren't copied into subclasses!

To look up a name in a class:

1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

```
>>> ch = CheckingAccount('Tom')  # Calls Account.__init__
>>> ch.interest     # Found in CheckingAccount
0.01
>>> ch.deposit(20)  # Found in Account
20
>>> ch.withdraw(5)  # Found in CheckingAccount
14
```

Object-Oriented Design

Designing for Inheritance

Don't repeat yourself; use existing implementations

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
```

Designing for Inheritance

Don't repeat yourself; use existing implementations

Attributes that have been overridden are still accessible via class objects

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
```

Attribute look-up on base class

Designing for Inheritance

Don't repeat yourself; use existing implementations

Attributes that have been overridden are still accessible via class objects

Look up attributes on instances whenever possible

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
```

Attribute look-up on base class

Preferred to CheckingAccount.withdraw_fee to allow for specialized accounts

Inheritance and Composition

- Composition: what one object has another object as an attribute

Inheritance is best for representing *is-a* relationships:

- CheckingAccount **is-a** specific type of account
- So, CheckingAccount inherits from Account

Composition is best for representing *has-a* relationships:

- E.g., a bank **has a** collection of bank accounts it manages
- So, A bank has a list of accounts as an attribute

Question

class Car

class Vehicle

class Toyota

class Engine

class Sound

class Plane

What relation each pair of classes has?

Is - a or has - a

Test question 1

```
class Fruit:
    pass    # do nothing


class Orange(Fruit):
    has_pulp = True

    def squeeze(self):
        return has_pulp


Orange().squeeze()
```

What is the output?

A: True

B: False

C: Error

Test Question 2

```
class Parent:
    def __init__(self, param):
        self.v1 = param

class Child(Parent):
    def __init__(self, param):
        self.v2 = param

obj = Child(11)
print(obj.v1 + " " + obj.v2)
```

What is the output?

A: None None

B: None 11

C: 11 None

D: 11 11

E: Error

```
class Mom:
    saying1 = "Wash your hands"

    def __init__(self, last_name):
        self.last_name = last_name

    def healthyDinner(self):
        return "Veggies and Protein"

class Dad:
    saying1 = "Can you climb even higher?"

    def healthyDinner(self):
        return "Protein and Sweets"

class Child(Dad, Mom):
    saying1 = "One more minute"
    last_name = "Pirate"
```

```
ch = Child("Smith")
print(ch.last_name)
print(ch.saying1)
print(ch.healthyDinner())
```

A: Smith, One more minute, Protein and Sweets

B: Pirate, One more minute, Protein and Sweets

C: Pirate, Can you climb..., Protein and Sweets

D: Smith, Can you climb..., Protein and Sweets

E: Something else

Multiple inheritance
(one child class has multiple parent classes)

Multiple Inheritance

```
class SavingsAccount(Account):
    deposit_fee = 2
    def deposit(self, amount):
        return Account.deposit(self, amount - self.deposit_fee)
```

A class may inherit from *multiple* base classes in Python

CleverBank marketing executive has an idea:

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

```
class SavingsAccount(Account):
    deposit_fee = 2
    def deposit(self, amount):
        return Account.deposit(self, amount - self.deposit_fee)
```

CleverBank marketing executive has an idea:

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

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

Multiple Inheritance

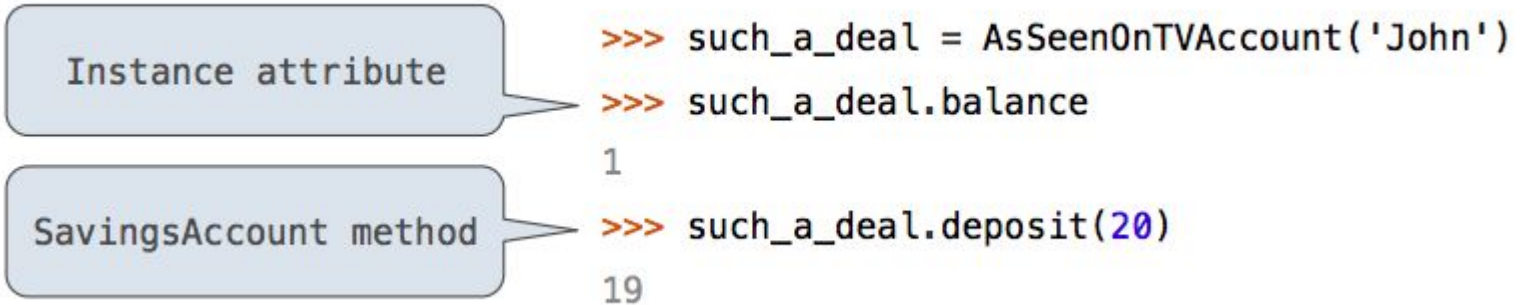
```
class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
    def __init__(self, account_holder):
        self.holder = account_holder
        self.balance = 1                    # A free dollar!
```

Instance attribute

```
>>> such_a_deal = AsSeenOnTVAccount('John')
>>> such_a_deal.balance
1
```

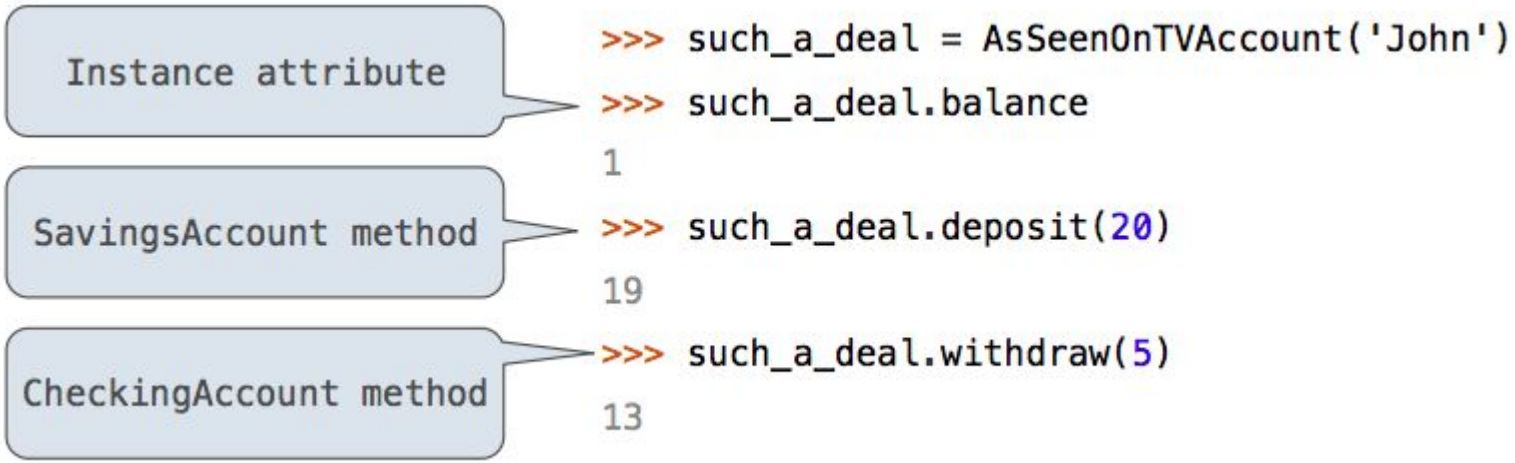
Multiple Inheritance

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

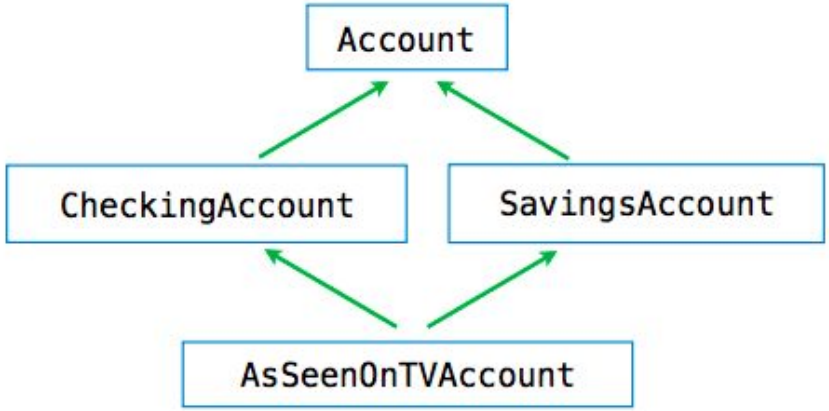


Multiple Inheritance

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



Resolving Ambiguous Class Attribute Names



Instance attribute

SavingsAccount method

CheckingAccount method

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

```
class A:
    z = -1
    def f(self, x):
        return B(x-1)

class B(A):
    n = 4
    def __init__(self, y):
        if y:
            self.z = self.f(y)
        else:
            self.z = C(y+1)


class C(B):
    def f(self, x):
        return x
```

```
>>> a = A()
>>> b = B(1)
>>> b.n = 5
>>> C(2).n
4
>>> C(2).z
2
>>> a.z == C.z
True
>>> a.z == b.z
False
>>> b.z.z.z
1
"""
```

```
class A:
    z = -1
    def f(self, x):
        return B(x-1)

class B(A):
    n = 4
    def __init__(self, y):
        if y:
            self.z = self.f(y)
        else:
            self.z = C(y+1)


class C(B):
    def f(self, x):
        return x
```

```
>>> a = A()
>>> b = B(1)
>>> b.n = 5


>>> C(2).n
???
>>> C(2).z
???
>>> a.z == C.z
???
>>> a.z == b.z
???
>>> b.z.z.z
???
"""
```