Inheritance

Slides adapted from Berkeley cs61a

- Attributes, and Attributes Lookup
- Attributes Assignments
- Inheritance
- Object-Oriented Design

  Inheritance vs. Composition vs. Mixin
- Multiple Inheritance
- Practice: *Attributes Lookup*

Review: class Account

demo_1:Account

Methods and Functions

Python distinguishes between:

- *Functions*, which we have been creating since the beginning of the course, and
- *Bound methods*, which couple together a function and the object on which that method will be invoked

Object + Function = Bound Method

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

>>> Account.deposit(tom_account, 1000)
1000
>>> tom_account.deposit(1020)
2020
```

Function: all arguments within parentheses

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

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

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

>>> type(Account)
<class 'type’>
```

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

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

>>> type(Account)
<class 'type’>
```

?

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

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

>>> type(Account)
<class 'type’>?
```

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

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

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

>>> type(Account)
<class 'type’>
```

?

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

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

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

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

>>> type(Account)
<class 'type’>
```

?

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

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

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

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

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

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

>>> type(Account)
<class 'type’>
```

?

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

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

my_object = MyClass()
MyClass = MetaClass()

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

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

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

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

type is the metaclass in Python

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

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

my_object = MyClass()
MyClass = MetaClass()

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

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

>>> type(Account)
<class 'type’>
```

?

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

type is the metaclass in Python

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

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

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

my_object = MyClass()
MyClass = MetaClass()

Terminology: Attributes, Functions, and Methods

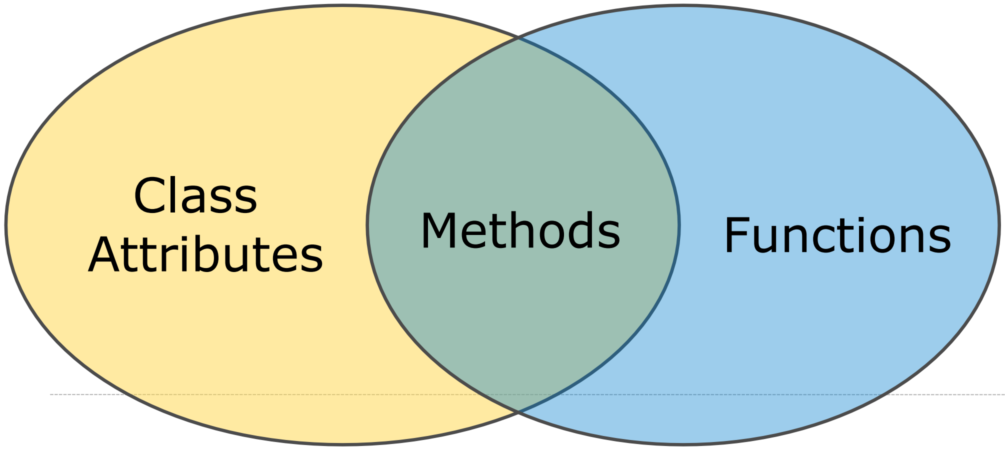
All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

Instance attribute: attribute of an instance

Class attribute: attribute of the class (of an instance)

Terminology:



Python object system:

Functions are (first-class) objects

Bound methods are also objects: a function that has its first parameter "self" already bound to an instance

Dot expressions evaluate to bound methods for class attributes that are functions

<instance>.<method_name>

Looking Up Attributes by Name

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

Looking Up Attributes by Name

`<expression> . <name>`

$$
\begin{array}{rcll}
\mathit{Expr} & ::= & \mathit{Name} & \text{-- Variable , see Rule 7.3} \\
& \mid & \mathit{Expr}(\langle \mathit{Expr} \rangle^*) & \text{-- Function call , see Rule 8.2} \\
& \mid & \mathit{Expr}.\mathit{Name} & \text{-- Attribute access , see Rule 9.10} \\
& \mid & \mathit{Expr}[\mathit{Expr}] & \text{-- Slice access , see Rule 13.11} \\
& \mid & \mathit{Expr}\ \mathit{BinOp}\ \mathit{Expr} & \text{-- Binary operator , see Rule 13.5} \\
& \mid & \mathit{UnaryOp}\ \mathit{Expr} & \text{-- Unary operator , see Rule 13.1} \\
& \mid & \texttt{yield}\ \mathit{Expr} & \text{-- Yield expression , see Rule 8.13} \\
& \mid & \mathit{Int} & \text{-- Literal integer , see Rule A.5} \\
& \mid & \mathit{Bool} & \text{-- Literal boolean , see Rule A.10} \\
& \mid & \mathit{String} & \text{-- Literal string , see Rule A.13} \\
& \mid & [\langle \mathit{Expr} \rangle^*] & \text{-- Literal list , see Rule A.21} \\
& \mid & (\langle \mathit{Expr} \rangle^*) & \text{-- Literal tuple , see Rule A.30} \\
& \mid & \{\langle \mathit{Expr} : \mathit{Expr} \rangle^*\} & \text{-- Literal dictionary, see Rule A.35}
\end{array}
$$

$$\mathit{BinOp} ::= + \mid - \mid * \mid / \mid \% \mid ** \mid // \mid == \mid != \mid < \mid <= \mid > \mid >= \mid \texttt{is} \mid \texttt{in} \mid \texttt{and} \mid \texttt{or}$$

$$\mathit{UnaryOp} ::= \texttt{not} \mid -$$

An executable operational semantics for Python, Gideon Smeding, Universiteit Utrecht, 2009

Looking Up Attributes by Name

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

To evaluate a dot expression:

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression

Looking Up Attributes by Name

<expression> . <name>

To evaluate a dot expression:

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression

2. <name> is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned

(demo: tom_account.balance)

Looking Up Attributes by Name

<expression> . <name>

To evaluate a dot expression:

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression
2. <name> is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned
3. If not, <name> is looked up in the class, which yields a class attribute value (if no such class attribute exists, an AttributeError is reported)

(demo: tom_account.interest, tom_account.noSuchAttribute)

Looking Up Attributes by Name

<expression> . <name>

To evaluate a dot expression:

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression

2. <name> is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned

3. If not, <name> is looked up in the class, which yields a class attribute value (if no such class attribute exists, an AttributeError is reported)

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

(demo: tom_account.deposit)

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

>>> 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 statements with a dot expression on their left-hand side affect attributes for the object of that dot expression (a.f = x)

- 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 (a.f = x)

- 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 (a.f = x)

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

Instance Attribute Assignment: tom_account.interest = 0.08

```
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 (a.f = x)

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

Instance Attribute Assignment: tom_account.interest = 0.08

This expression evaluates to an object

```
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 (a.f = x)

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

Instance Attribute Assignment: tom_account.interest = 0.08

This expression evaluates to an object

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

```
class Account:
    interest = 0.02
    def_init_(self, holder):
        self.holder = holder
        self.balance = 0
    ...
tom_account = Account('Tom')
```

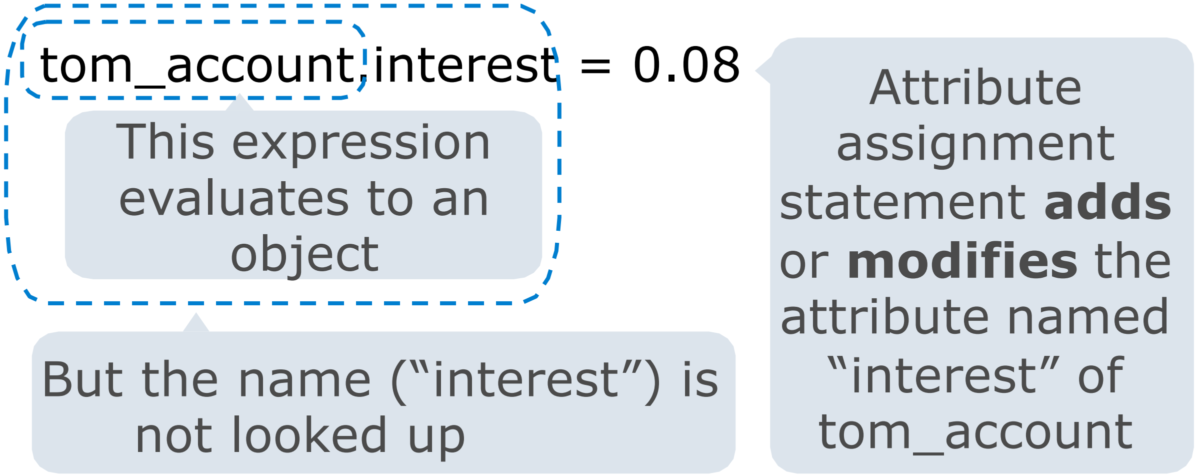
Assignment to Attributes

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression (a.f = x)

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

Instance Attribute Assignment:

```
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

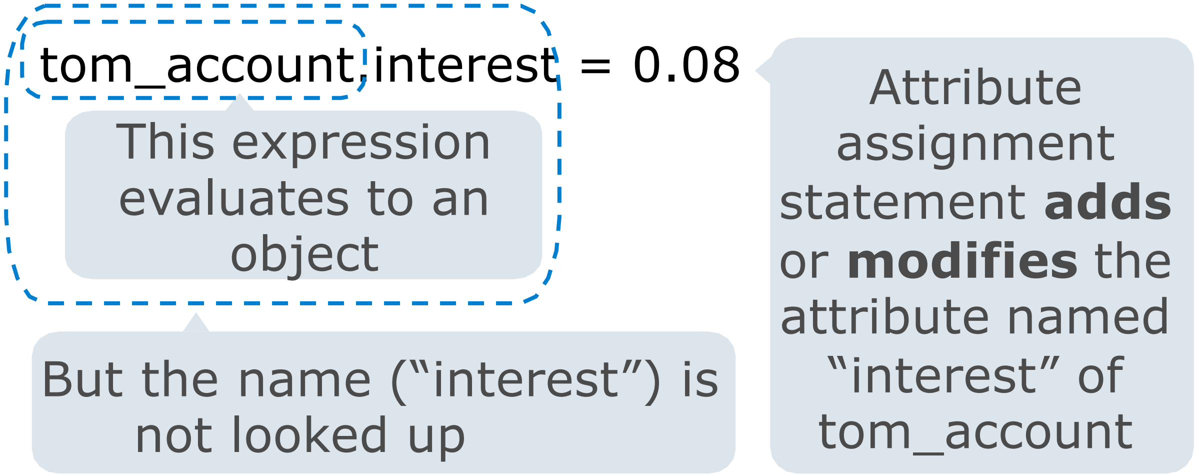
Attribute assignment statement **adds** or **modifies** the attribute named “interest” of tom_account

Assignment to Attributes

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression (a.f = x)

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

Instance 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 Account:
    interest = 0.02
    def_init_(self, holder):
        self.holder = holder
        self.balance = 0
    ...
tom_account = Account('Tom')
```

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_)

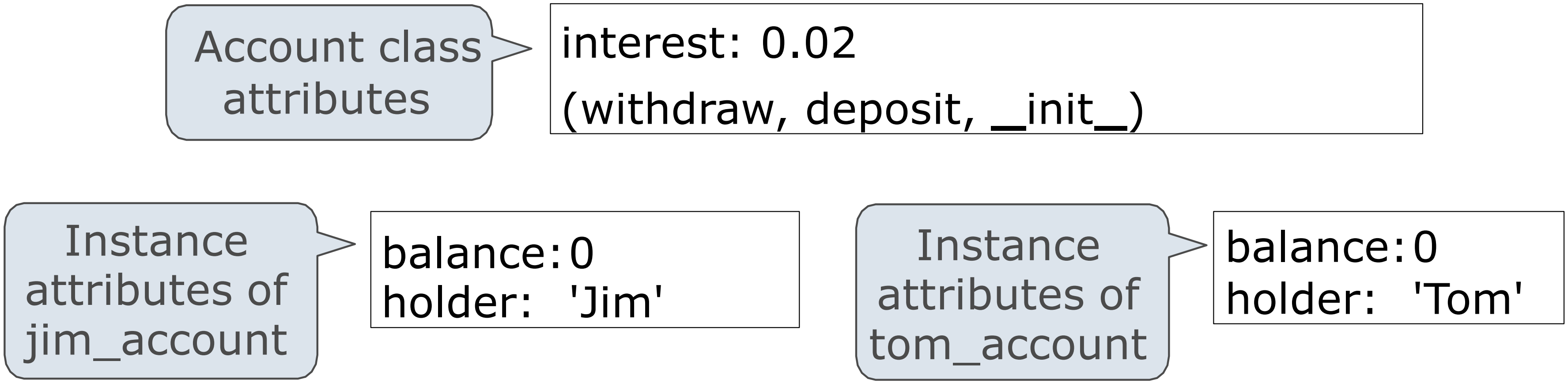
Instance attributes of jim_account

balance: 0
holder: 'Jim'

Instance attributes of tom_account

balance: 0
holder: 'Tom'

Attribute Assignment Statements



Account class attributes

interest: 0.02

(withdraw, deposit, _init_)

Instance attributes of jim_account

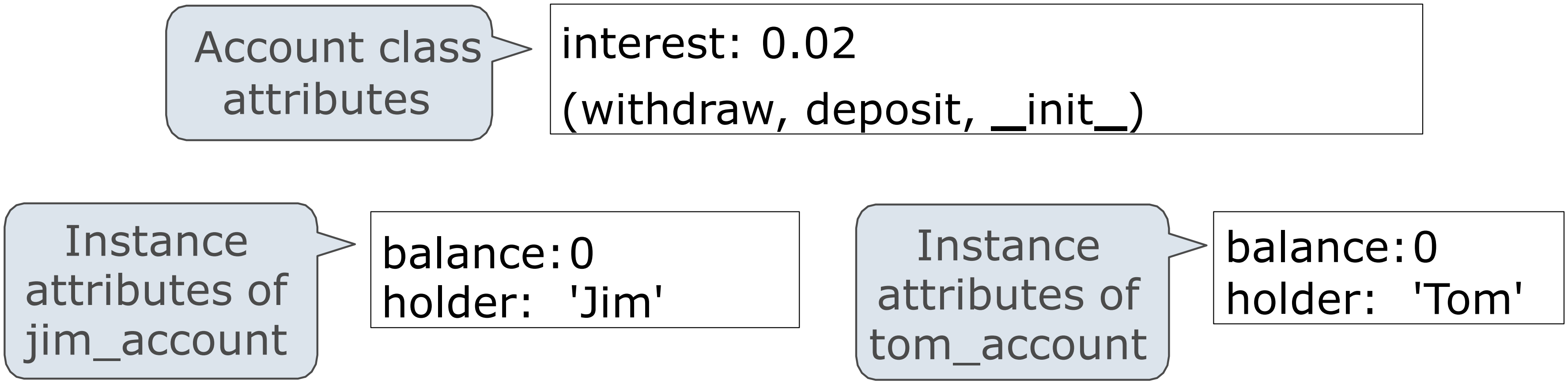
balance: 0
holder: 'Jim'

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

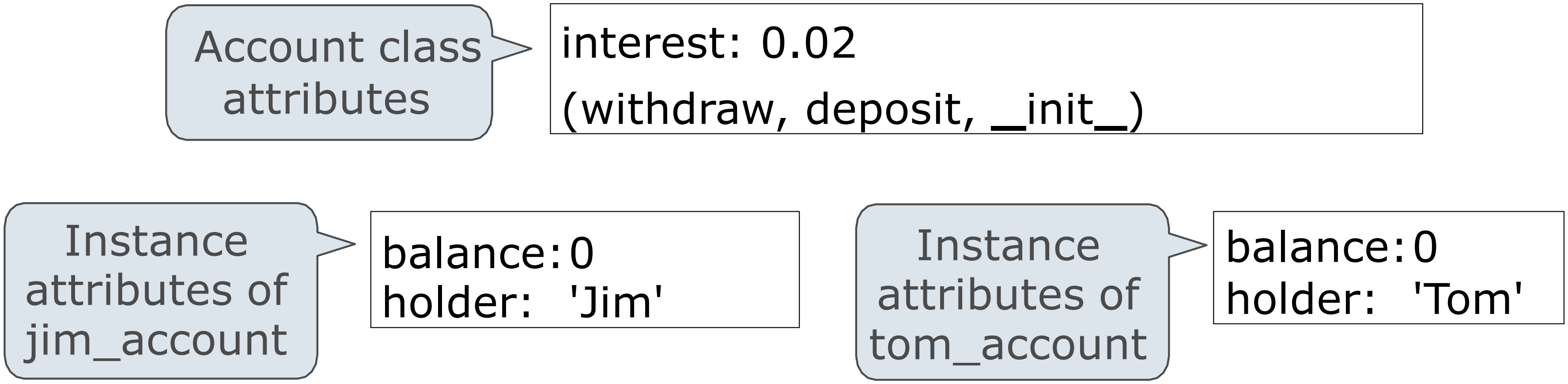
balance: 0
holder: 'Jim'

Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

>>> jim_account.interest

```

Attribute Assignment Statements



Account class attributes

interest: 0.02
(withdraw, deposit, _init_)

Instance attributes of jim_account

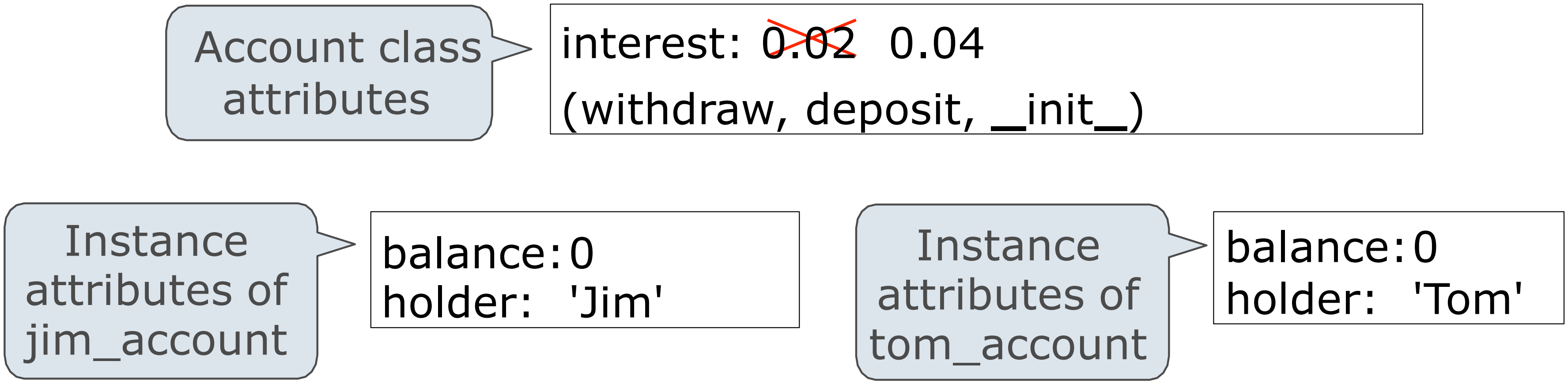
balance: 0
holder: 'Jim'

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~~ 0.04

(withdraw, deposit, _init_)

Instance attributes of jim_account

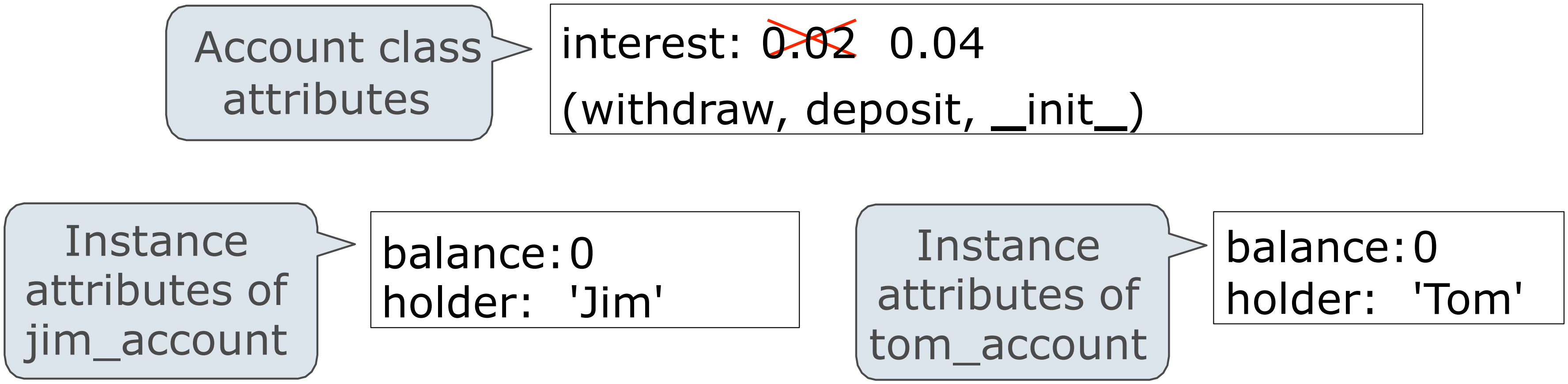
balance: 0
holder: 'Jim'

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

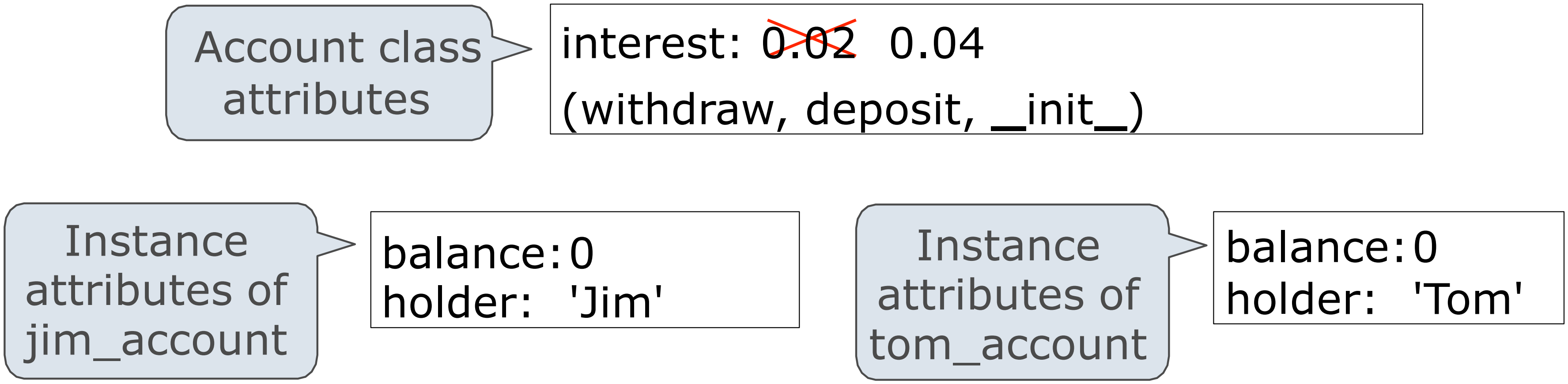
balance: 0
holder: 'Jim'

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
```

Attribute Assignment Statements



Account class attributes

interest: ~~0.02~~ 0.04

(withdraw, deposit, _init_)

Instance attributes of jim_account

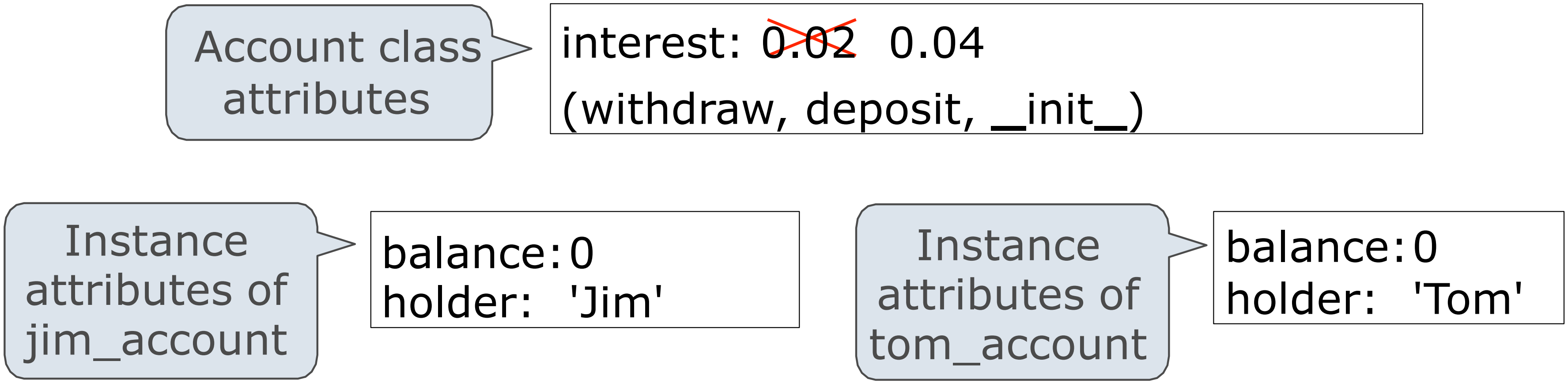
balance: 0
holder: 'Jim'

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

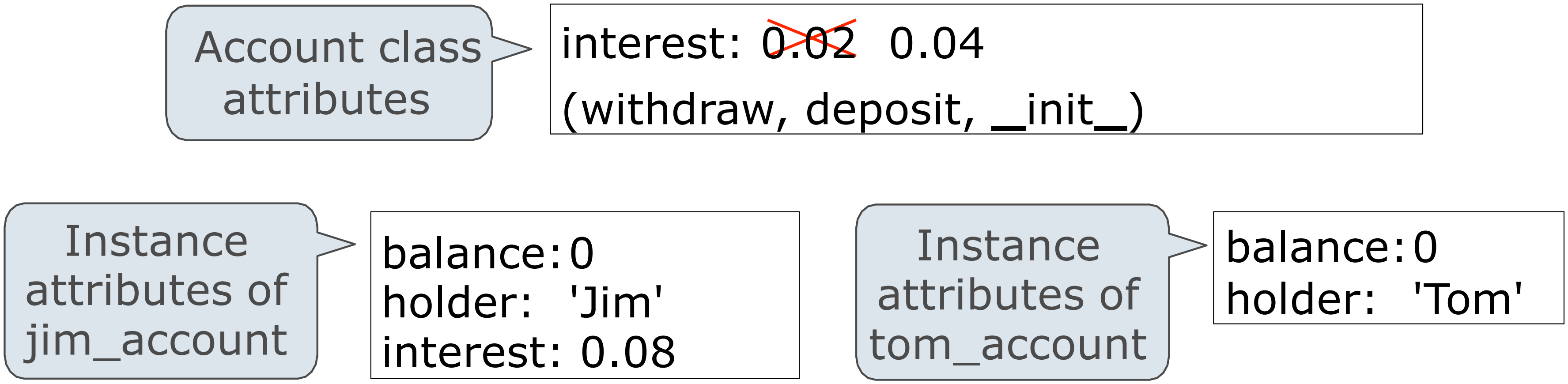
balance: 0
holder: 'Jim'

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'
interest: 0.08

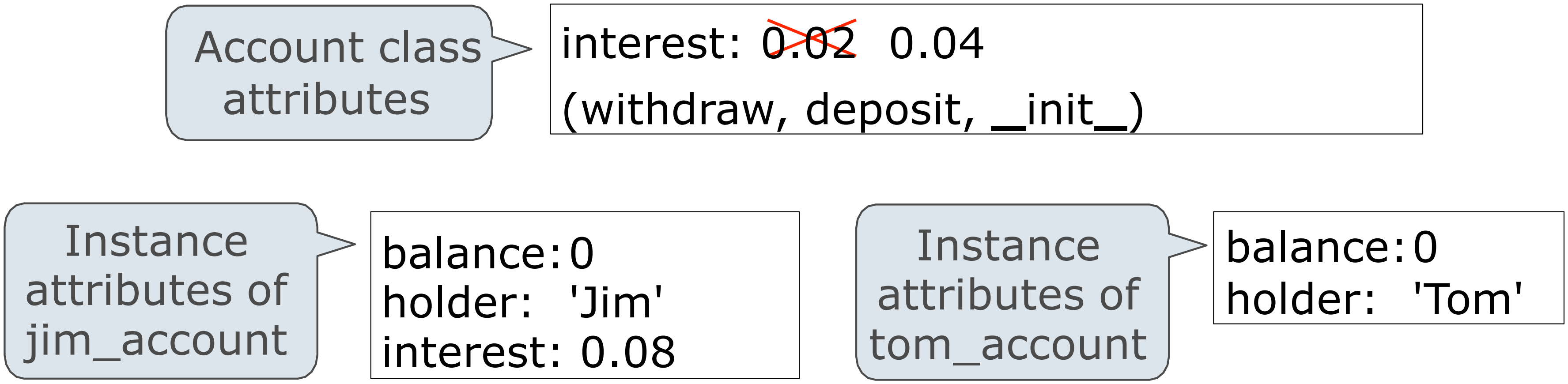
Instance attributes of tom_account

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

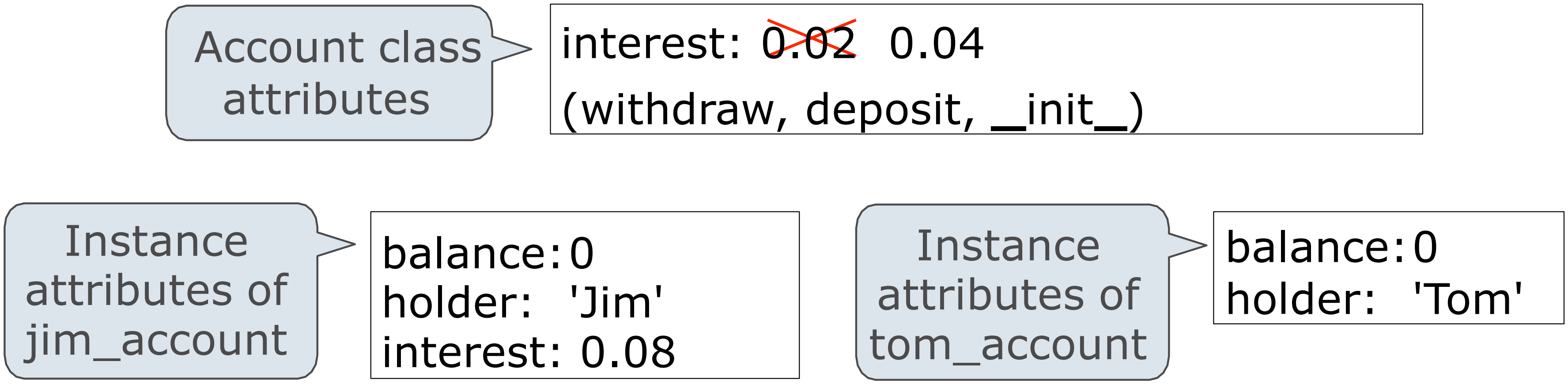
Instance attributes of tom_account

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
```

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

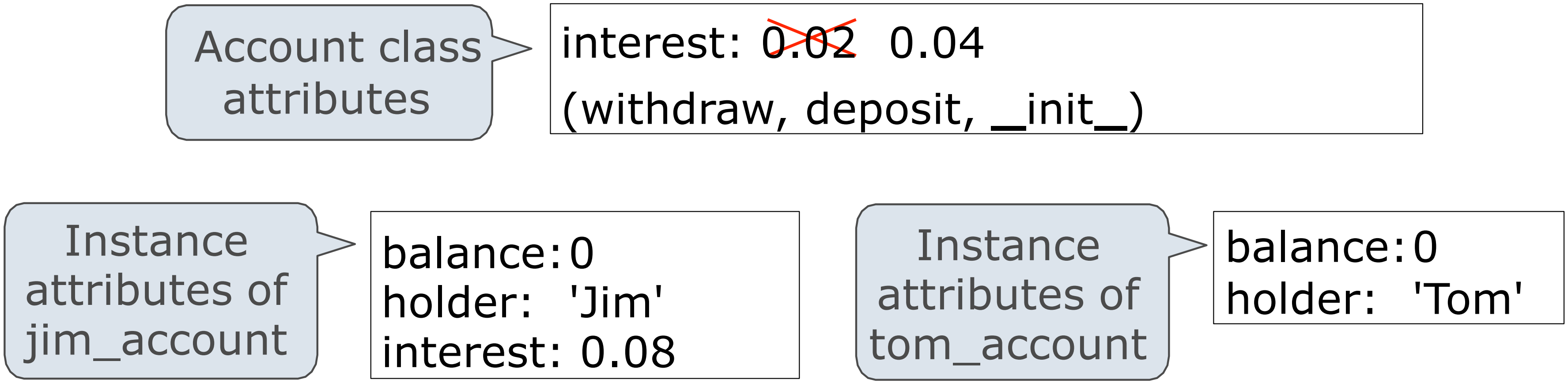
Instance attributes of tom_account

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

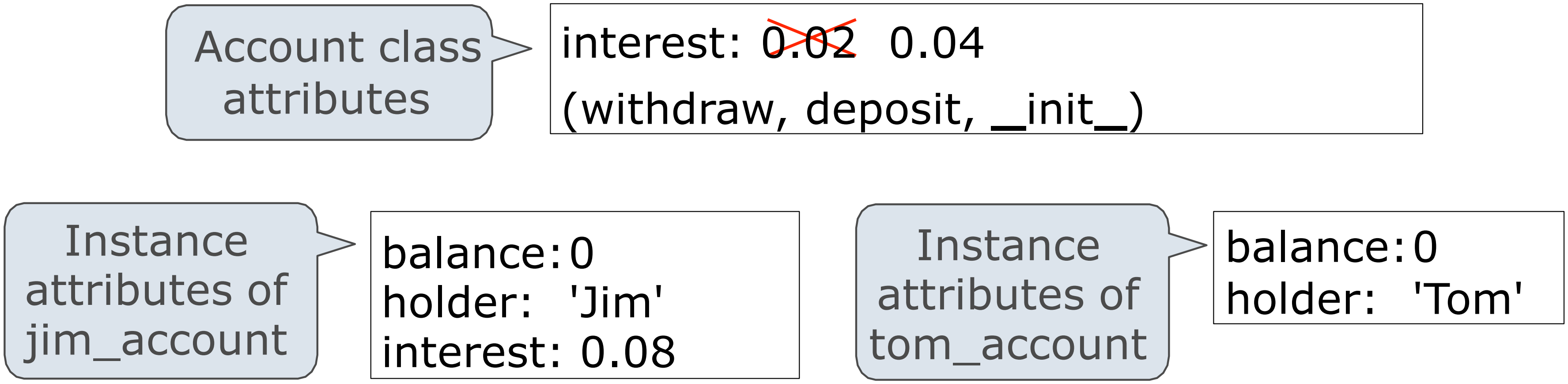
balance: 0
holder: 'Tom'

```
>>> 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
```

[redacted]

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

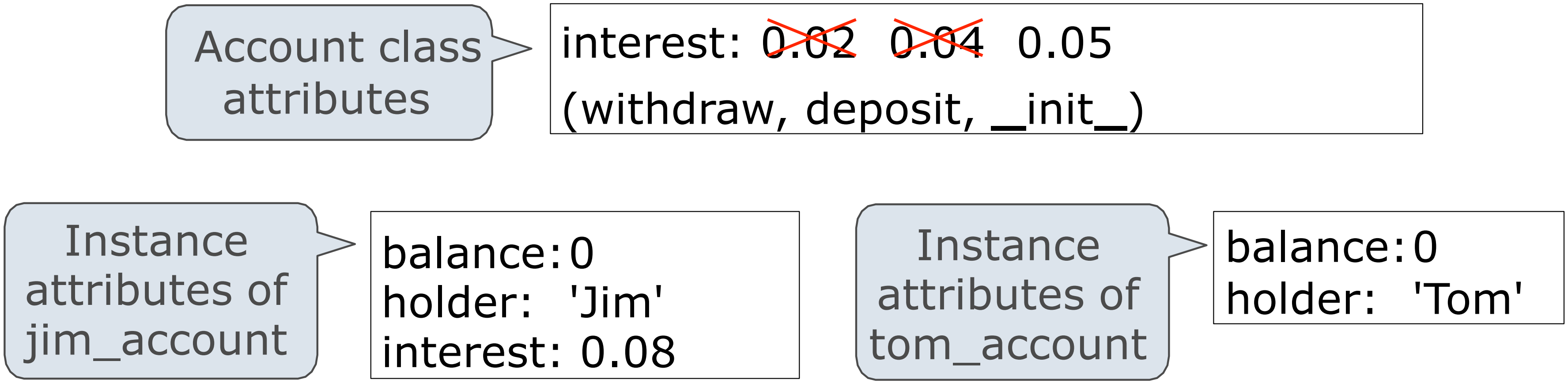
Instance attributes of tom_account

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~~ 0.05
(withdraw, deposit, _init_)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

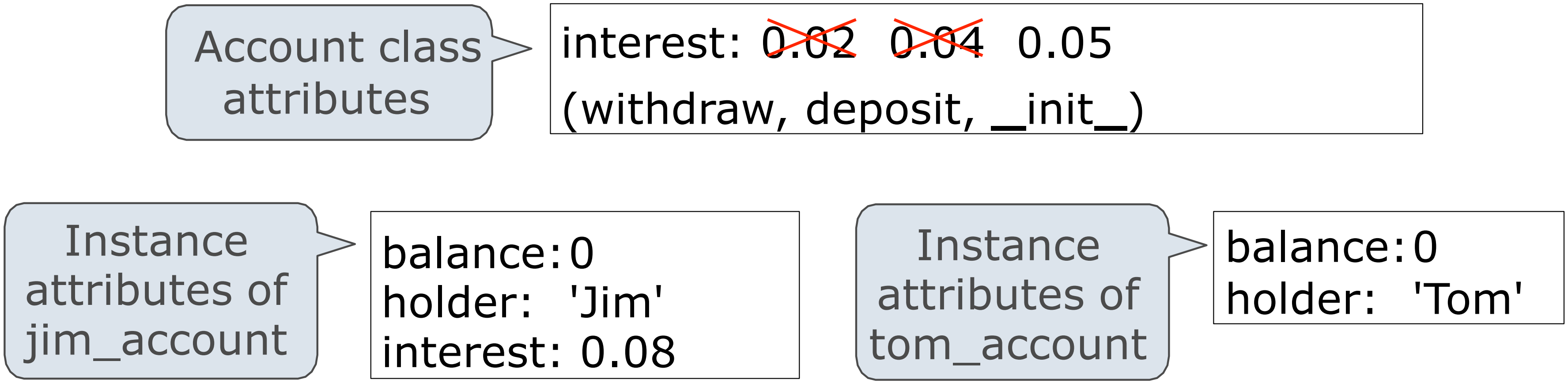
Instance attributes of tom_account

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

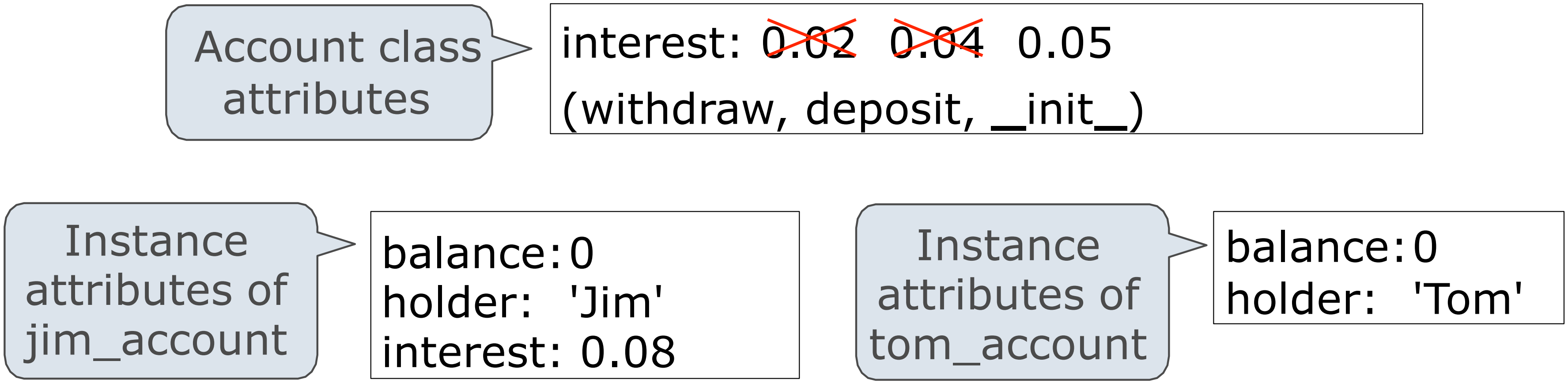
Instance attributes of tom_account

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
```

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

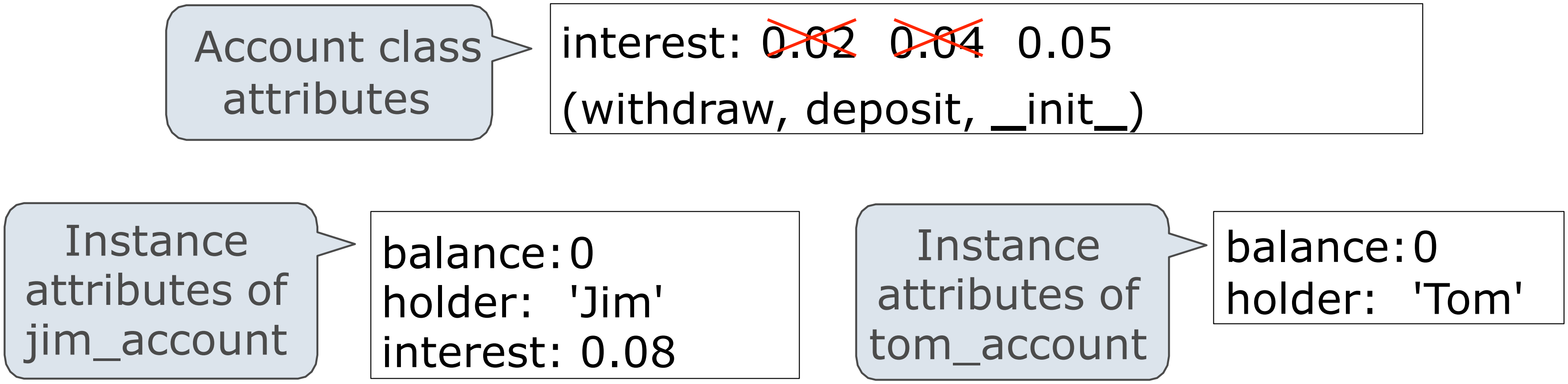
Instance attributes of tom_account

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

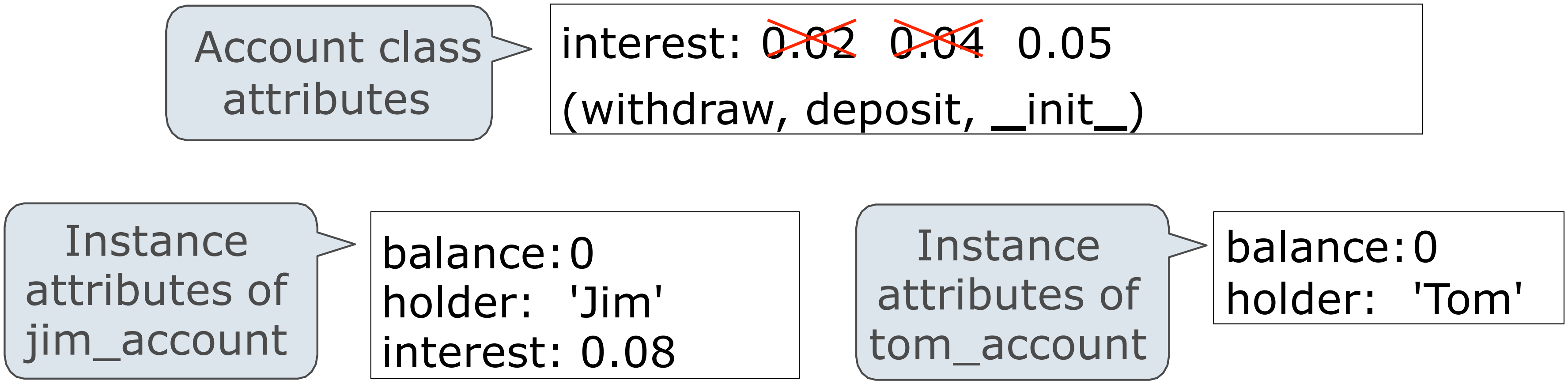
Instance attributes of tom_account

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
```

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
```

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>
```

Conceptually, the new subclass inherits attributes of its base 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
>>> 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

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

Looking Up Attribute Names on Classes

Base class attributes *aren't* copied into subclasses!

To look up a name in a class:

1. If it names an attribute in the class, return the attribute value.

2. Otherwise, look up the name in the base class, if there is one.

```
>>> 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
```

demo_2: CheckingAccount

Object-Oriented Design

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
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
            return Account.withdraw(self, amount + self.withdraw_fee)
```

Assume in the future, a subclass **GreenCheckingAccount** whose interest is 0.01 but withdraw_fee is only 0.23

Attribute look-up on base class

Preferred to CheckingAccount.withdraw_fee to allow for specialized accounts

demo_3: Bank

Inheritance: Use It Carefully

Inheritance helps code reuse but NOT for code reuse!

Inheritance: Use It Carefully

Inheritance helps code reuse but NOT for code reuse!

Disadvantages of inheritance

- Breaks encapsulation

  Inheritance forces the developer of the subclass to know about the internals of the superclass

  e.g., override HashSet’s add and addAll

Inheritance: Use It Carefully

Inheritance helps code reuse but NOT for code reuse!

Disadvantages of inheritance

- Breaks encapsulation

  Inheritance forces the developer of the subclass to know about the internals of the superclass

  e.g., override HashSet’s add and addAll

- Unnecessary cost for inheritance maintenance

  e.g., the cost of superclasses’ fields storage, constructors invocation, while only few behaviors of superclasses are needed

Composition

Colloquially, composition means

> "If you want to reuse some behavior, put that behavior in a class, create an object of that class, include it as an attribute, and call its methods when the behavior is needed"

Composition

Colloquially, composition means

> "If you want to reuse some behavior, put that behavior in a class, create an object of that class, include it as an attribute, and call its methods when the behavior is needed"

- Composition does not break encapsulation, and does not affect the types (all public interfaces remain unchanged)

Composition

Colloquially, composition means

“If you want to reuse some behavior, put that behavior in a class, create an object of that class, include it as an attribute, and call its methods when the behavior is needed”

- Composition does not break encapsulation, and does not affect the types (all public interfaces remain unchanged)
- No need to involve in possibly complex hierarchy, and easy to understand and implement

Inheritance vs. Composition

Guidance to choose inheritance or composition

- By conceptual difference
- By practical need

Inheritance vs. Composition

Guidance to choose inheritance or composition

- By conceptual difference

  Inheritance represents “is-a” relationship

  e.g., a checking account is a specific type of account

  Composition represents “has-a” relationship

  e.g., a bank has a collection of bank accounts it manages

- By practical need

Inheritance vs. Composition

Guidance to choose inheritance or composition

- By conceptual difference

  Inheritance represents “is-a” relationship

  e.g., a checking account is a specific type of account

  Composition represents “has-a” relationship

  e.g., a bank has a collection of bank accounts it manages

- By practical need

  If type B wants to expose all public methods of type A (B can be used wherever A is expected), favors inheritance

  If type B needs only parts of behaviors exposed by type A, favors composition

Inheritance vs. Composition

Implementing composition means we need to wrap the delegation logic (delegated to the composed object) into certain methods, in which case inheritance’s “direct reuse” seems more convenient.

Inheritance vs. Composition

Do we have some approach to somewhat take the advantages of both inheritance and composition?

Implementing composition means we need to wrap the delegation logic (delegated to the composed object) into certain methods, in which case inheritance’s “direct reuse” seems more convenient.

Mixin

Mixin is a class that contains methods for use by other classes without having to be the parent class of those other classes, and without having to use delegation to a composed object.

Mixin

Mixin is a class that contains methods for use by other classes without having to be the parent class of those other classes, and without having to use delegation to a composed object.

- Mixin is usually considered as “included” rather than “inherited”

Mixin

Mixin is a class that contains methods for use by other classes without having to be the parent class of those other classes, and without having to use delegation to a composed object.

- Mixin is usually considered as “included” rather than “inherited”
- But unlike composition (as also an “included” approach), the mixin-ed methods appear to be inherited (no object delegation)

Mixin

Mixin is a class that contains methods for use by other classes without having to be the parent class of those other classes, and without having to use delegation to a composed object.

- Mixin is usually considered as “included” rather than “inherited”
- But unlike composition (as also an “included” approach), the mixin-ed methods appear to be inherited (no object delegation)
- Unlike Python, some languages such as Ruby and Scala, has language support to enforce the syntax/semantics of Mixin
  - E.g., Mixin is called module in Ruby, and trait in Scala

Mixin

Mixin is a class that contains methods for use by other classes without having to be the parent class of those other classes, and without having to use delegation to a composed object.

- Mixin is usually considered as “included” rather than “inherited”
- But unlike composition (as also an “included” approach), the mixin-ed methods appear to be inherited (no object delegation)
- Unlike Python, some languages such as Ruby and Scala, has language support to enforce the syntax/semantics of Mixin
  - E.g., Mixin is called module in Ruby, and trait in Scala
- Mixin is usually considered as a mean for multiple inheritance

Multiple Inheritance

Multiple Inheritance

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

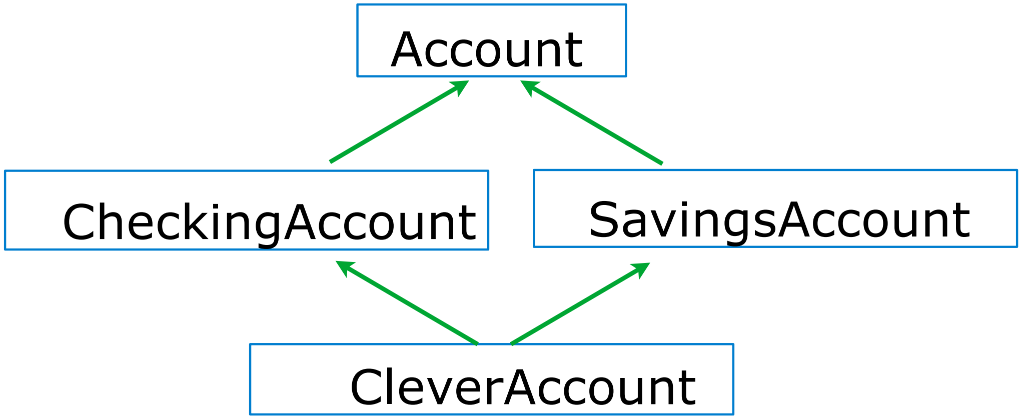
A class may inherit from multiple base classes in Python

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

Multiple Inheritance



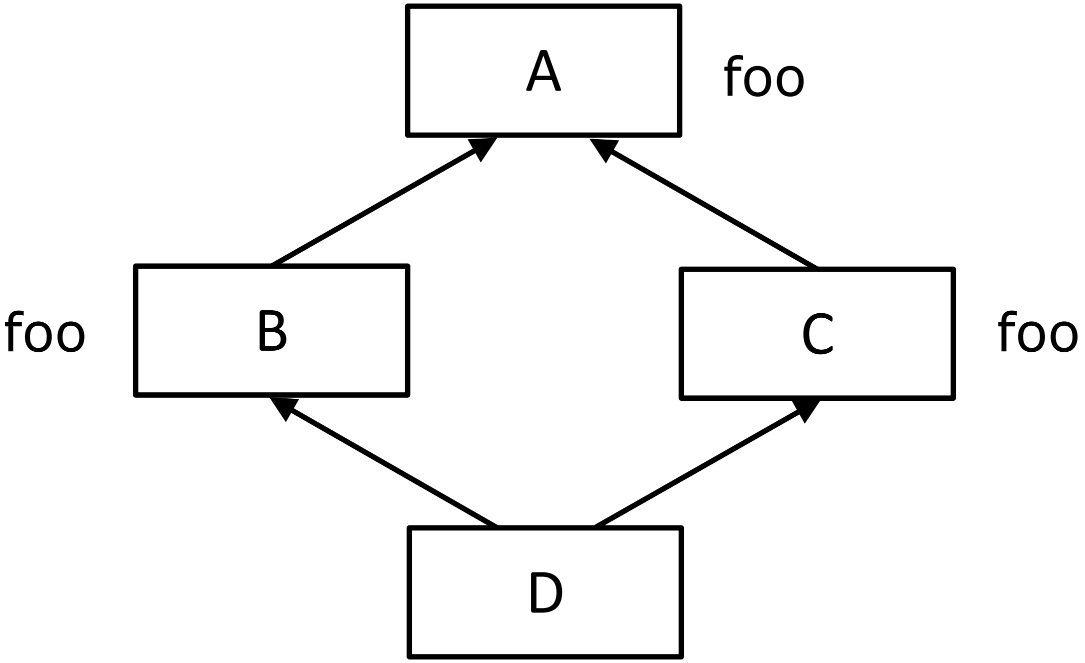
Instance attribute

SavingsAccount method

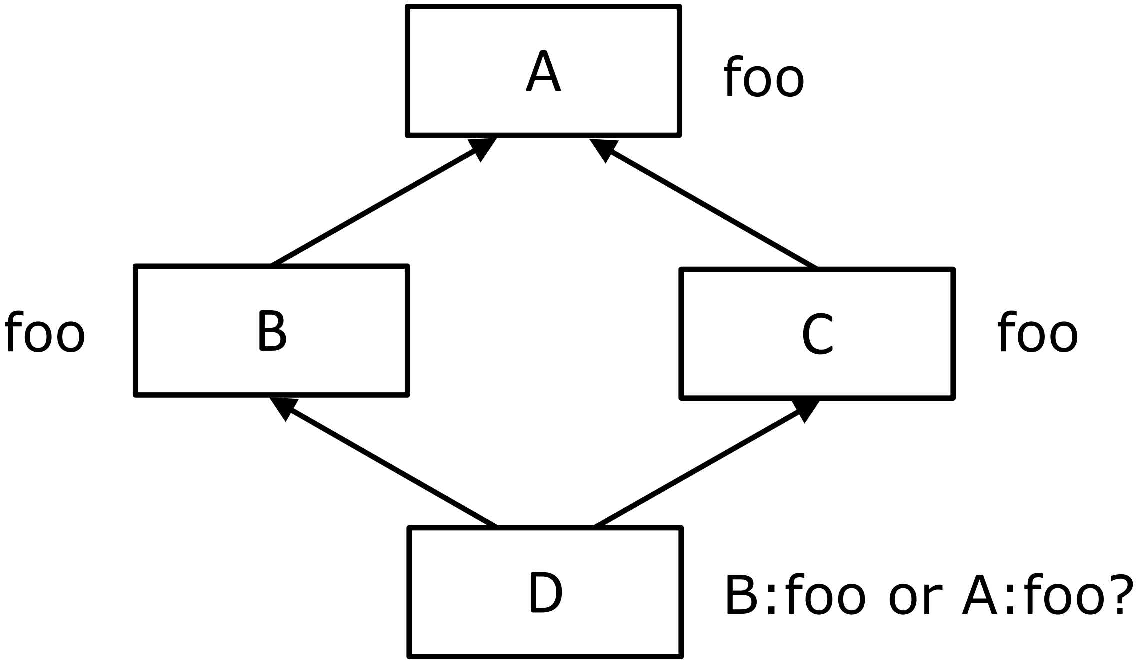
CheckingAccount method

```
>>> tom= CleverAccount(‘Tom')
>>> tom.balance
1
>>> tom.deposit(20)
19
>>> tom.withdraw(5)
13
```

Diamond Problem



Diamond Problem



A

foo

foo

B

C

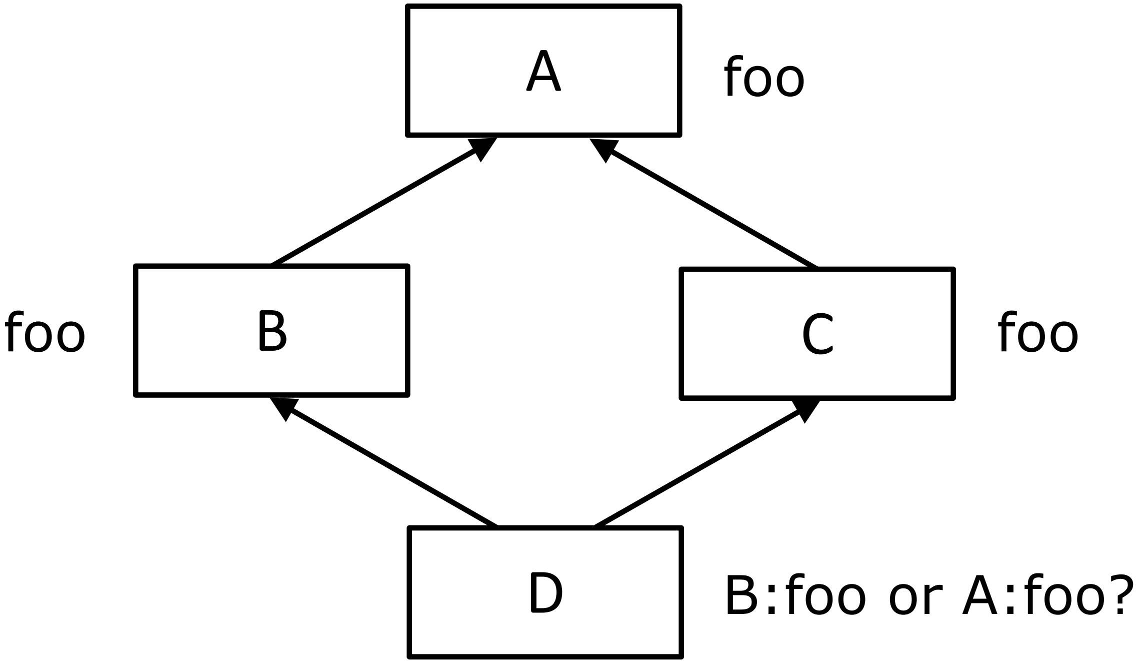
foo

D

B:foo or A:foo?

Method Resolution Order (MRO)

Diamond Problem



Method Resolution Order (MRO)

C3 Linearization algorithm for method resolution while doing multiple inheritance

Practice: Attributes Lookup

Inheritance and Attribute Lookup

```
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
```

```
>>> a.z == C.z
```

```
>>> a.z == b.z
```

Which evaluates to an integer?

b.z
b.z.z
b. z. z. z
b.z.z.z.z
None of these

Inheritance and Attribute Lookup

```
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
```

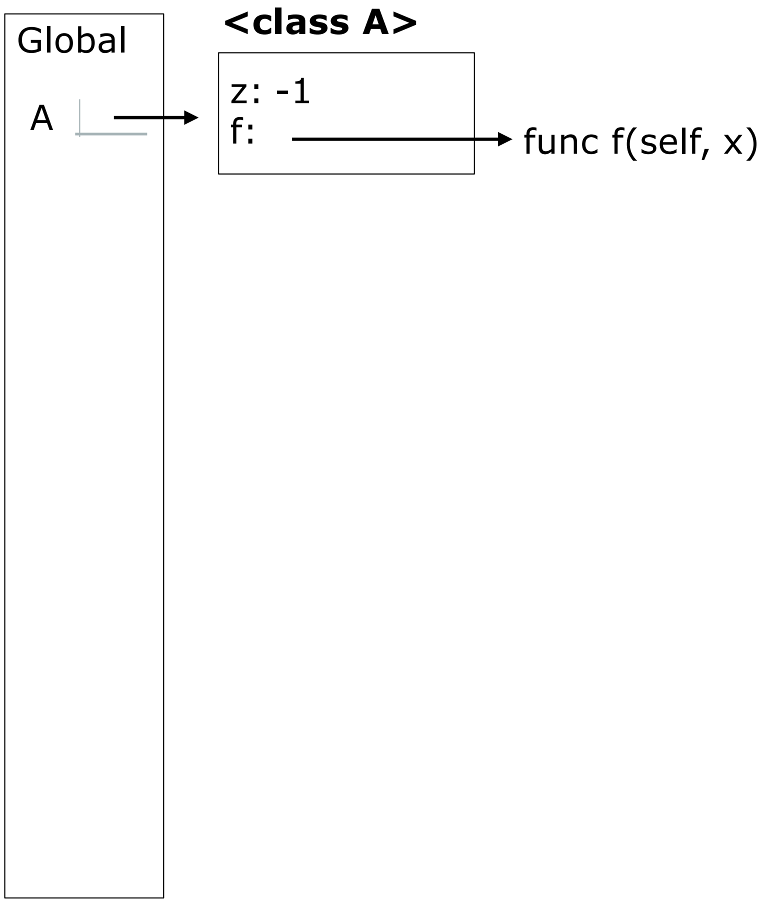
```
>>> C(2).n
```

```
>>> a.z == C.z
```

```
>>> a.z == b.z
```

Which evaluates to an integer?

- b.z
- b.z.z
- b.z.z.z
- b.z.z.z.z
- None of these



Inheritance and Attribute Lookup

```
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
```

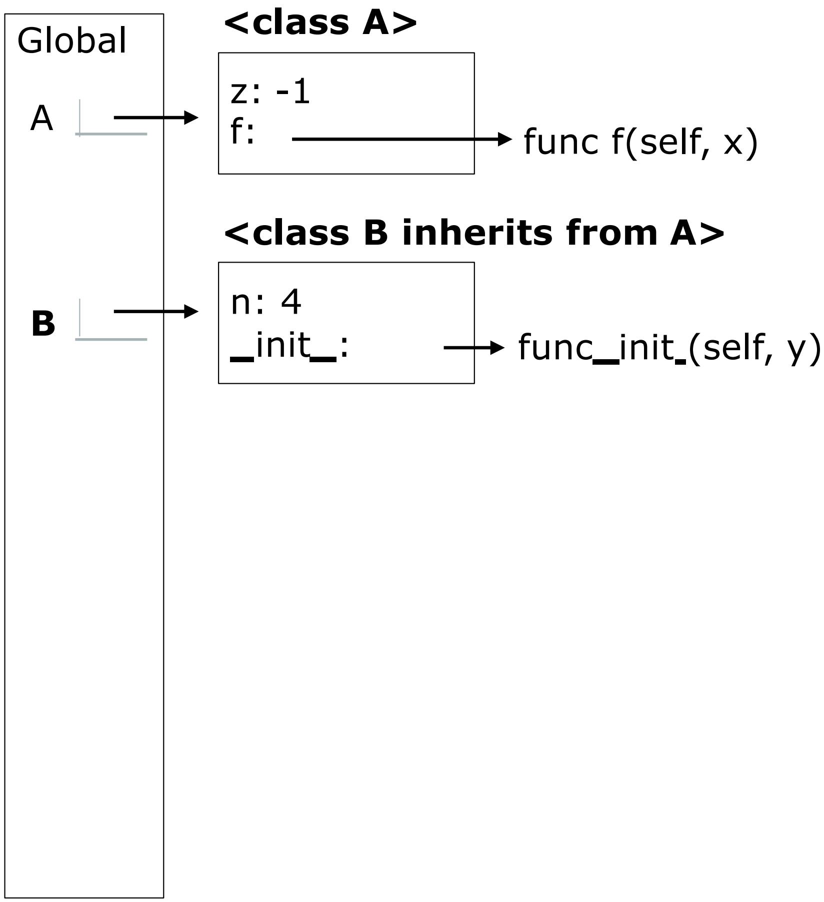
```
>>> C(2).n
```

```
>>> a.z == C.z
```

```
>>> a.z == b.z
```

Which evaluates to an integer?

- b.z
- b.z.z
- b.z.z.z
- b.z.z.z.z
- None of these



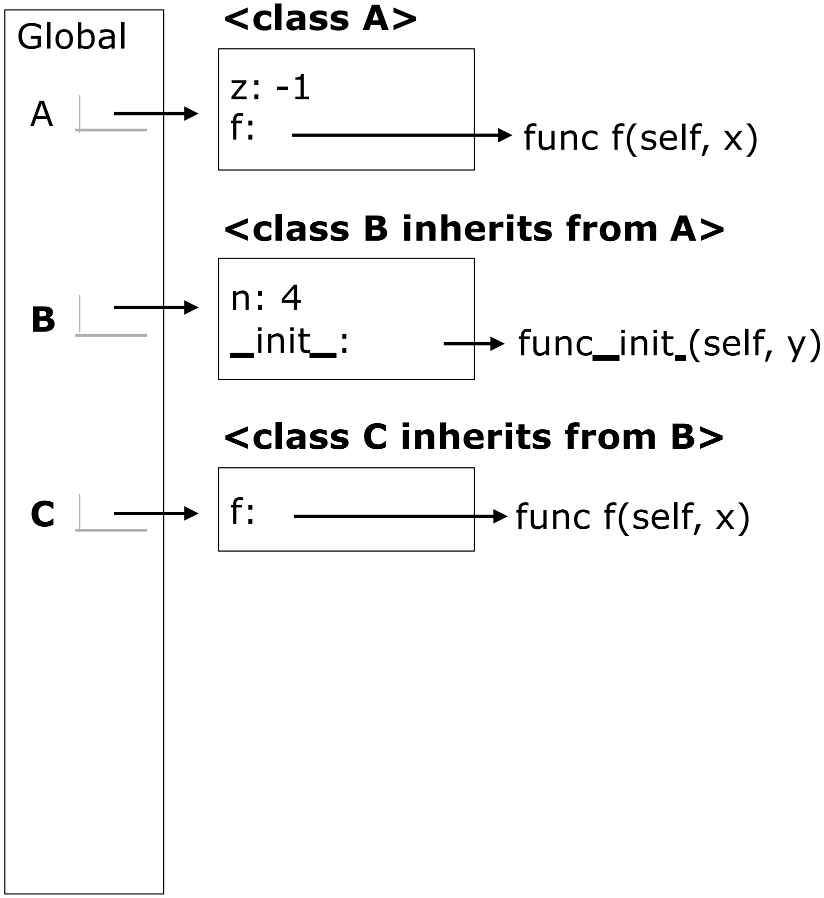
Inheritance and Attribute Lookup

```
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


>>> a.z == C.z


>>> a.z == b.z
```

Which evaluates to an integer?

- b.z
- b.z.z
- b.z.z.z
- b.z.z.z.z
- None of these



Inheritance and Attribute Lookup

```
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
```

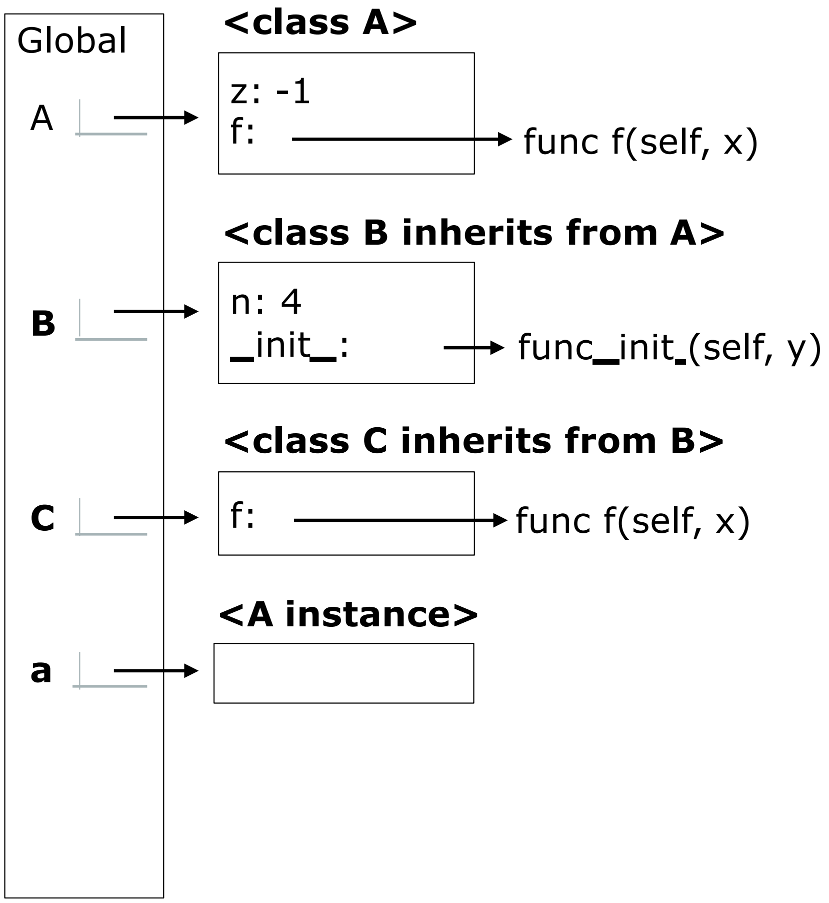
```
>>> C(2).n
```

```
>>> a.z == C.z
```

```
>>> a.z == b.z
```

Which evaluates to an integer?

- b.z
- b.z.z
- b.z.z.z
- b.z.z.z.z
- None of these



Global

A

B

C

a

<class A>

z: -1

f:

func f(self, x)

<class B inherits from A>

n: 4

__init__:

func __init__(self, y)

<class C inherits from B>

f:

func f(self, x)

<A instance>

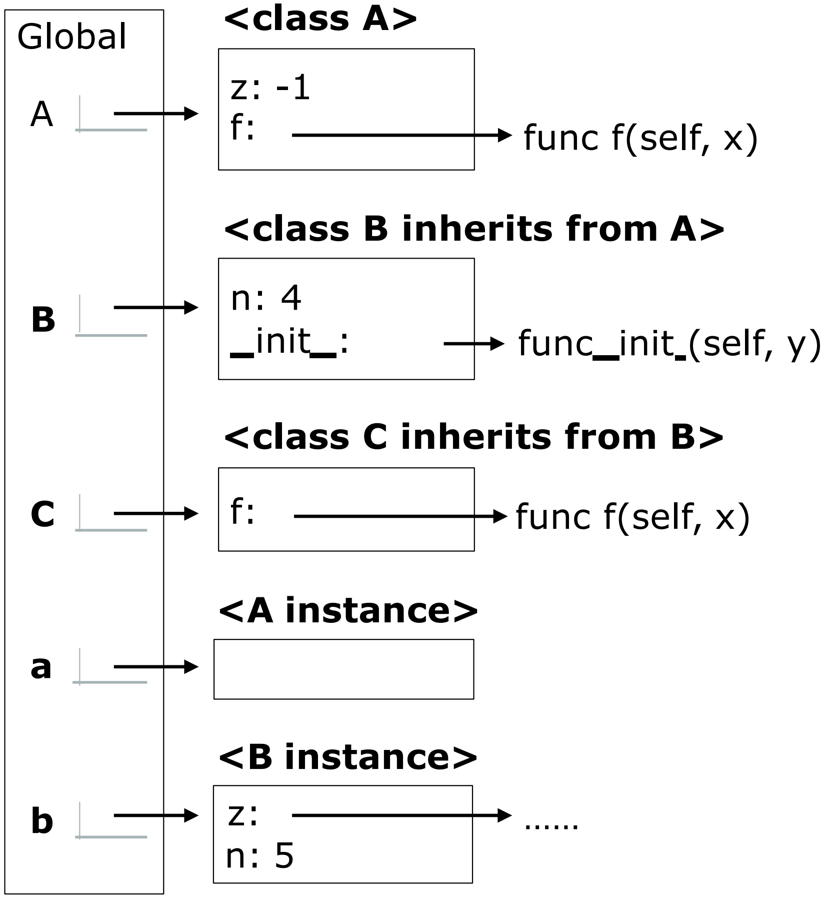
Inheritance and Attribute Lookup

```
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


>>> a.z == C.z


>>> a.z == b.z
```

Which evaluates to an integer?

- b.z
- b.z.z
- b.z.z.z
- b.z.z.z.z
- None of these



Global

A

B

C

a

b

<class A>

z: -1

f: → func f(self, x)

<class B inherits from A>

n: 4

__init__: → func__init_(self, y)

<class C inherits from B>

f: → func f(self, x)

<A instance>

<B instance>

z: → ......

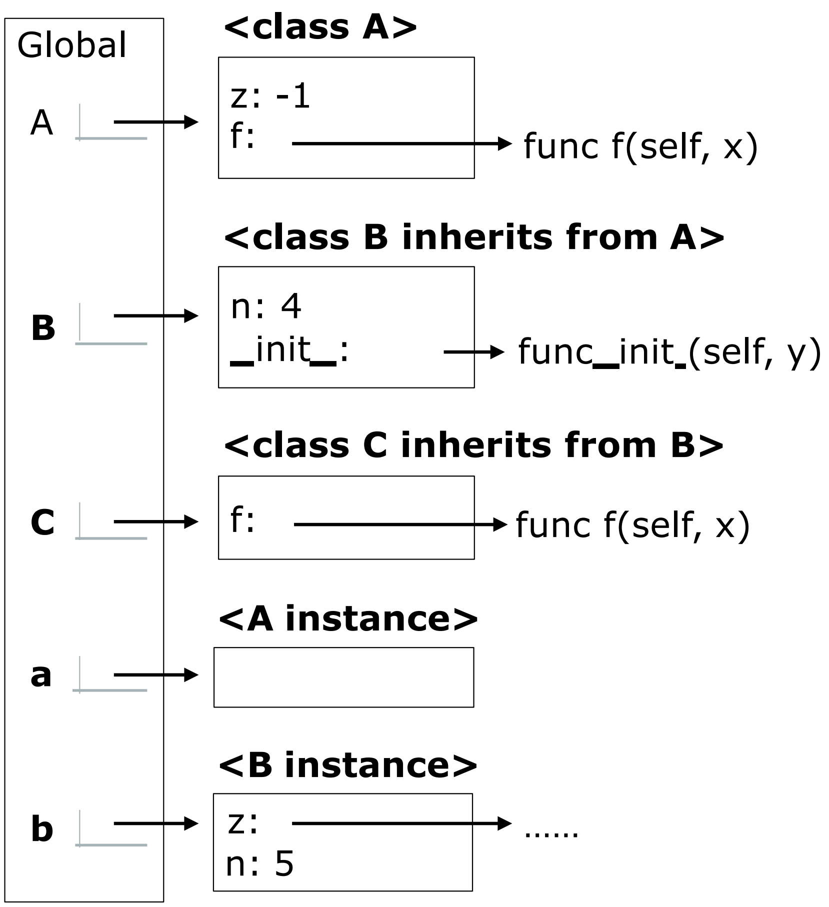
n: 5

Inheritance and Attribute Lookup

```
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

>>> a.z == C.z



>>> a.z == b.z
```

Which evaluates to an integer?
- b.z
- b.z.z
- b.z.z.z
- b.z.z.z.z
- None of these



Global

A

B

C

a

b

<class A>

z: -1
f:

func f(self, x)

<class B inherits from A>

n: 4
__init__:

func__init_(self, y)

<class C inherits from B>

f:

func f(self, x)

<A instance>

<B instance>

z:
n: 5

......

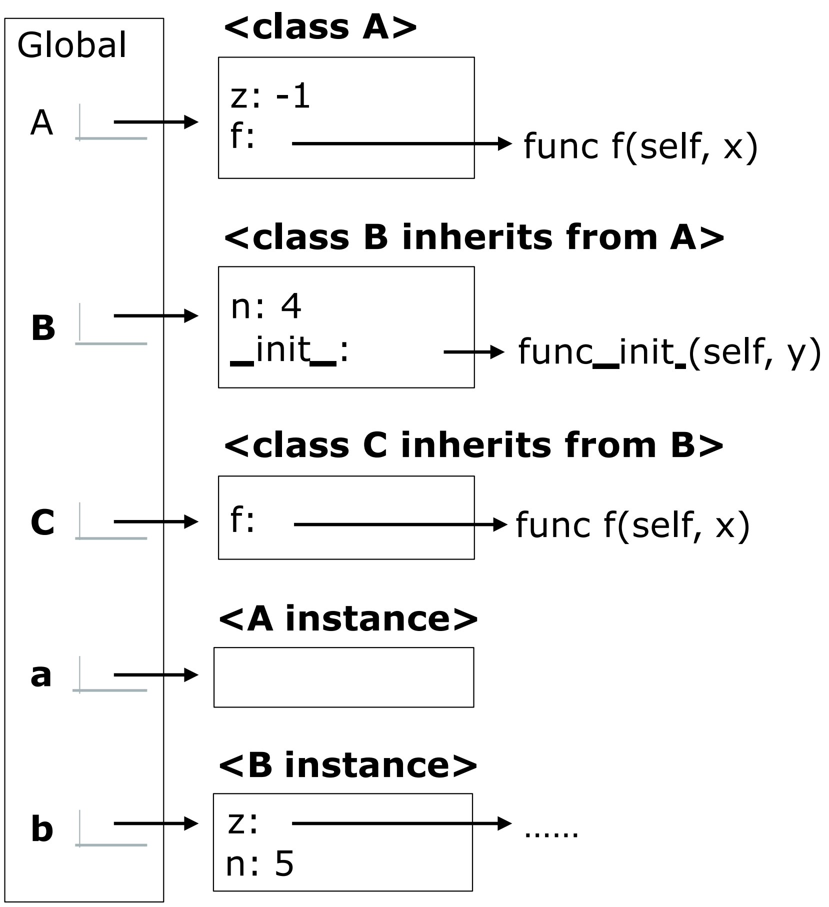
Inheritance and Attribute Lookup

```
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

>>> a.z == C.z

    True

>>> a.z == b.z
```

Which evaluates to an integer?

b.z
b.z.z
b.z.z.z
b.z.z.z.z
None of these



Global

A

B

C

a

b

<class A>

z: -1
f: → func f(self, x)

<class B inherits from A>

n: 4
__init__: → func __init__(self, y)

<class C inherits from B>

f: → func f(self, x)

<A instance>

<B instance>

z: → ......
n: 5

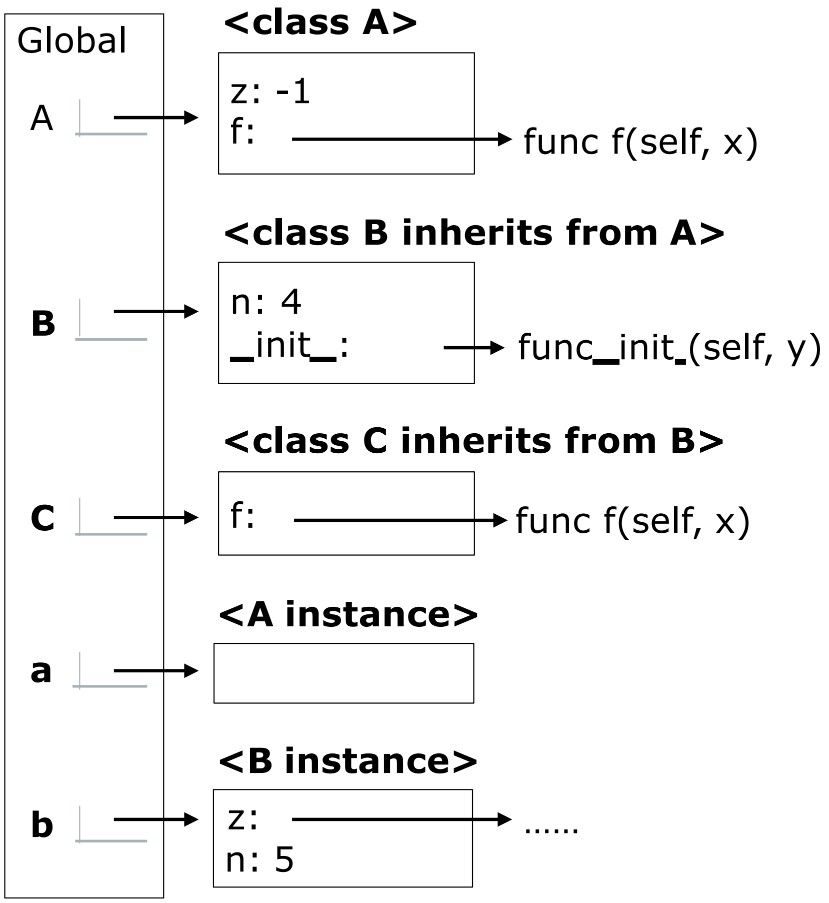
Inheritance and Attribute Lookup

```
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

>>> a.z == C.z

    True

>>> a.z == b.z

    False
```

Which evaluates to an integer?

- b.z
- b.z.z
- b.z.z.z
- b.z.z.z.z
- None of these



Global

A

B

C

a

b

<class A>

z: -1

f: func f(self, x)

<class B inherits from A>

n: 4

__init__: func__init_(self, y)

<class C inherits from B>

f: func f(self, x)

<A instance>

<B instance>

z: ......

n: 5

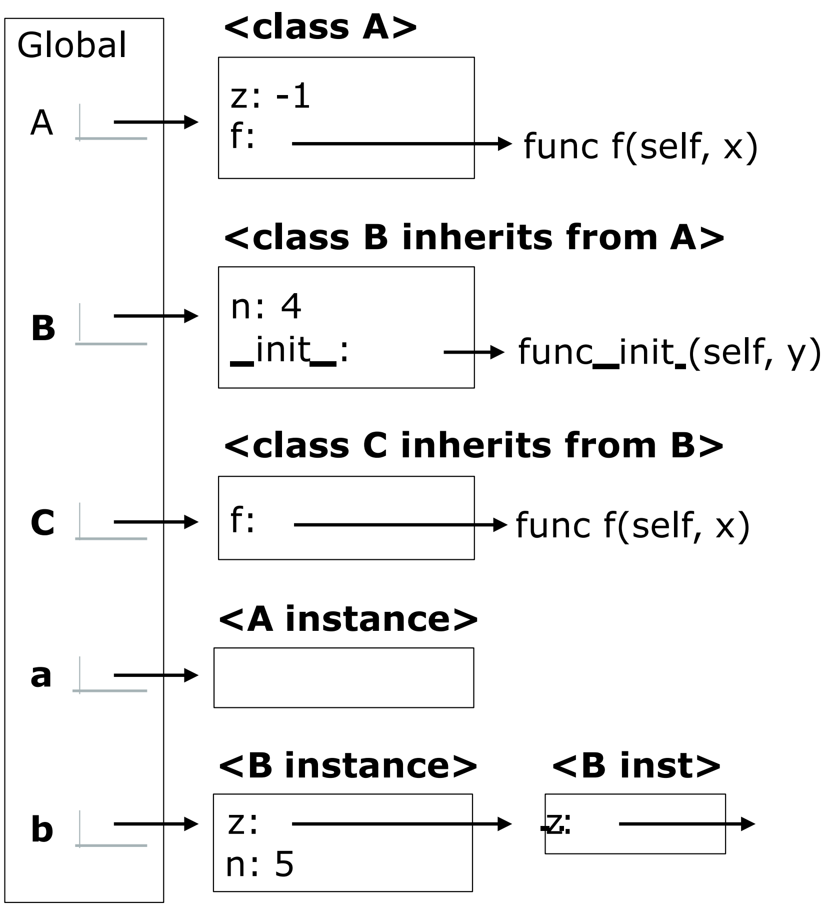
Inheritance and Attribute Lookup

```
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

>>> a.z == C.z

    True

>>> a.z == b.z

    False
```

Which evaluates to an integer?

- b.z
- b.z.z
- b.z.z.z
- b.z.z.z.z
- None of these



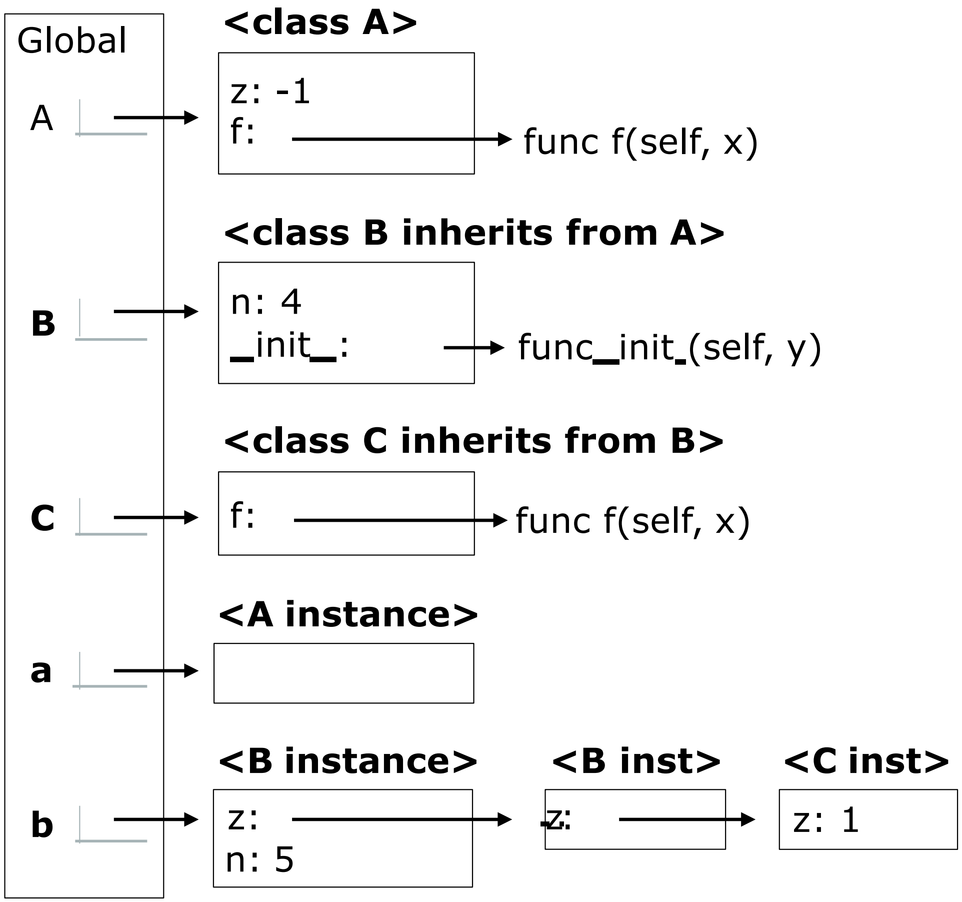
Inheritance and Attribute Lookup

```
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

>>> a.z == C.z

    True

>>> a.z == b.z

    False
```

Which evaluates to an integer?

- b.z
- b.z.z
- ▶ b.z.z.z
- b.z.z.z.z
- None of these



The X You Need To Understand In This Lecture

- Instance attribute vs. class attribute
- Rules of attribute lookup
- Rules of attribute assignment
- Rules of inheritance
- Difference between inheritance, composition and mixin