Inheritance

Slides adapted from Berkeley cs61a

- Attributes Assignments
- Inheritance
- Object-Oriented Design

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

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 a is an instance, then assignment sets an instance attribute
- If a 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 a is an instance, then assignment sets an instance attribute
- If a 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 a is an instance, then assignment sets an instance attribute
- If a 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 a is an instance, then assignment sets an instance attribute
- If a 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 a is an instance, then assignment sets an instance attribute
- If a 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 a is an instance, then assignment sets an instance attribute
- If a 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')
```

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 a is an instance, then assignment sets an instance attribute
- If a 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

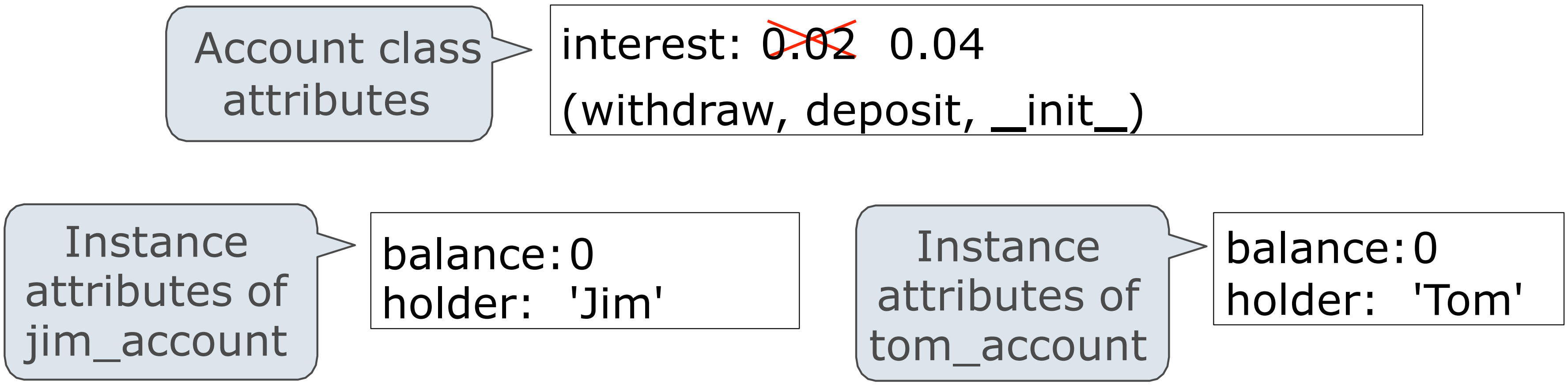
balance: 0
holder: 'Jim'

Instance attributes of tom_account

balance: 0
holder: 'Tom'

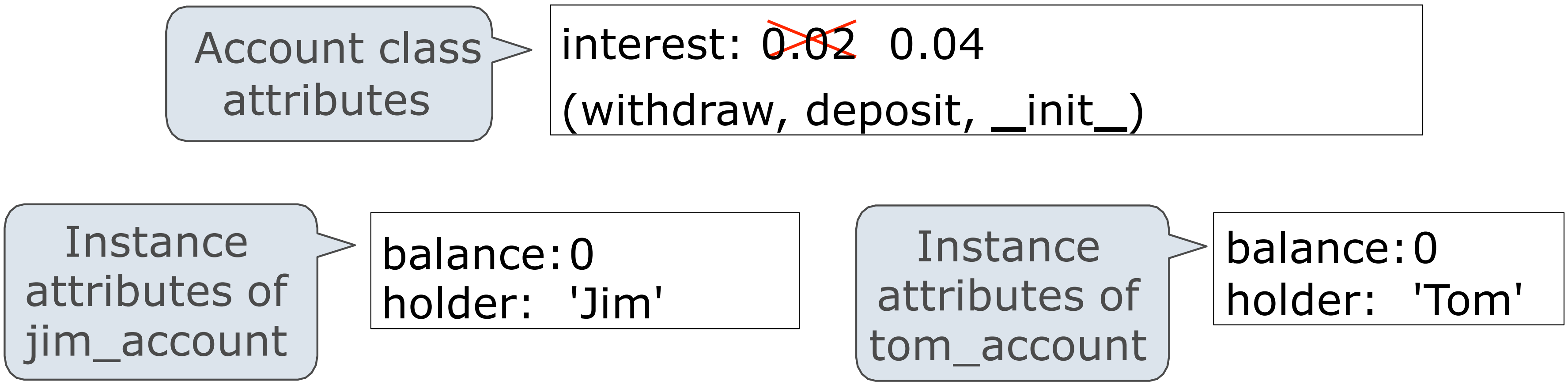
```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
```

Attribute Assignment Statements




```
>>> 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




```
>>> 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
```

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

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

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

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

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
```

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.

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
```

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 (Account)

Look up attributes on instances whenever possible

Designing for Inheritance

Don't repeat yourself; use existing implementations

Attributes that have been overridden are still accessible via class objects (Account)

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

Designing for Inheritance

Don't repeat yourself; use existing implementations

Attributes that have been overridden are still accessible via class objects (Account)

Look up attributes on instances whenever possible

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

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

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

demo_3: Bank

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

CleverAccount 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

Multiple Inheritance

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

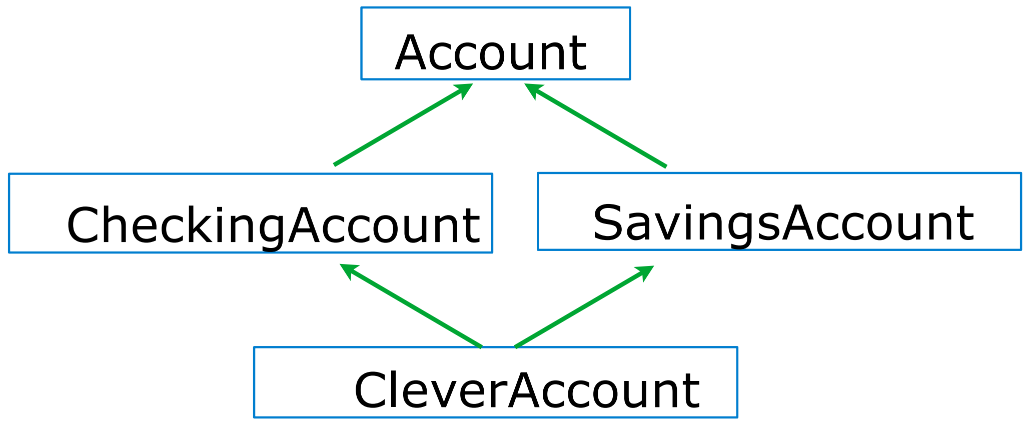
A class may inherit from multiple base classes in Python

CleverAccount 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



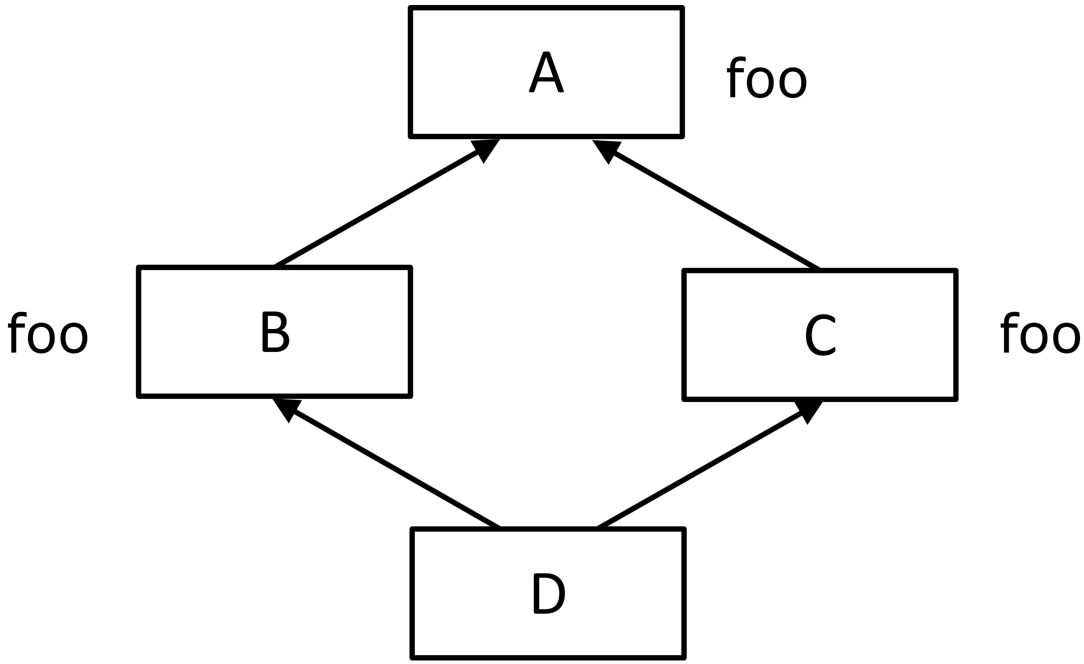

```
>>> tom= CleverAccount('Tom')
>>> tom.balance
1
>>> tom.deposit(20)
18
>>> tom.withdraw(5)
12
```

Instance attribute (tom.balance)

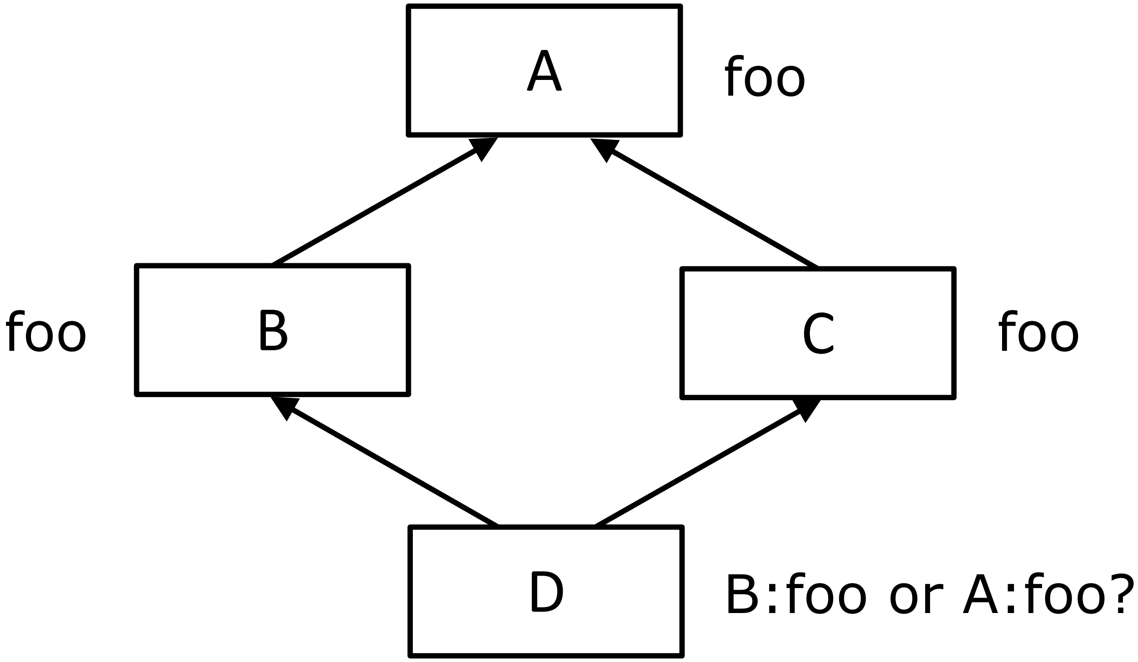
SavingsAccount method (tom.deposit)

CheckingAccount method (tom.withdraw)

Diamond Problem

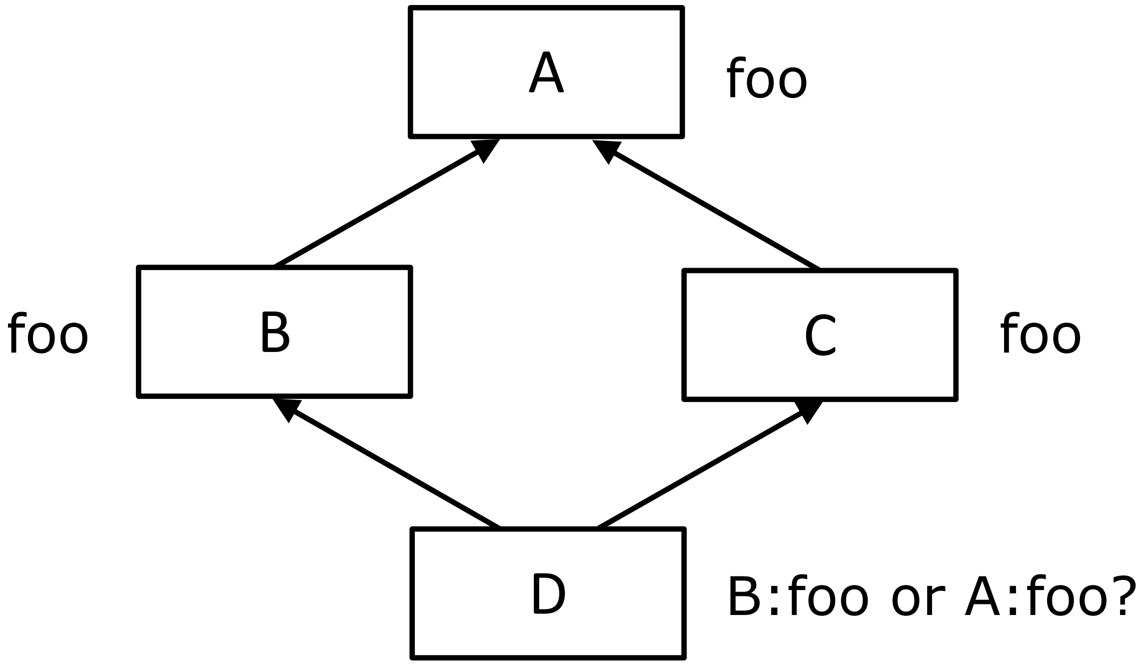


Diamond Problem



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

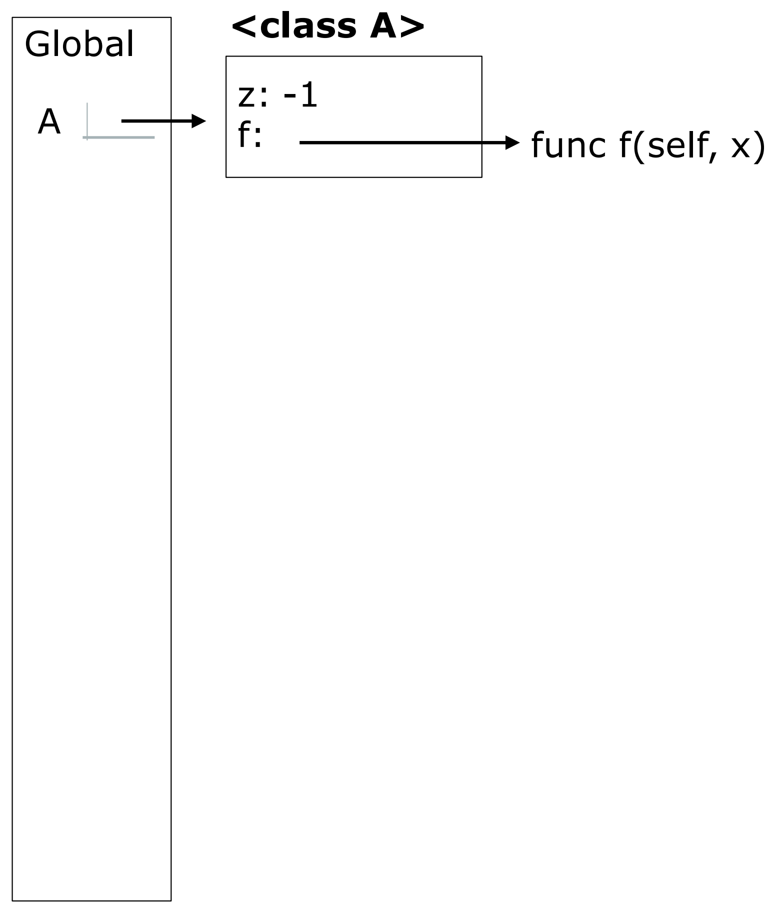
Inheritance and Attribute Lookup

```
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



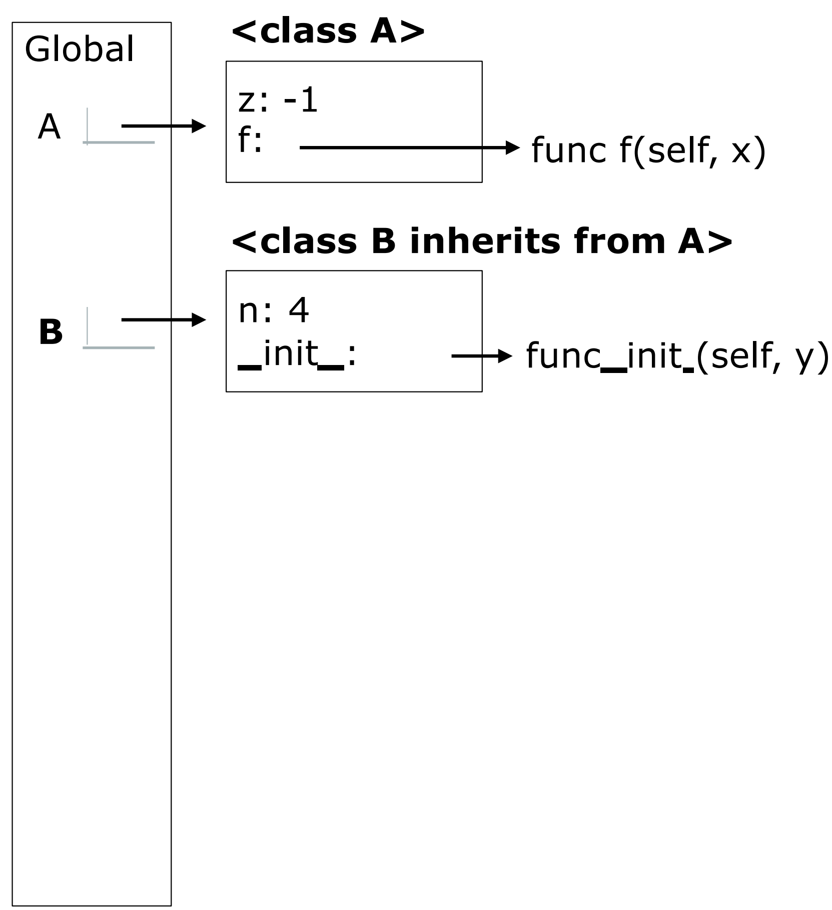
Inheritance and Attribute Lookup

```
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



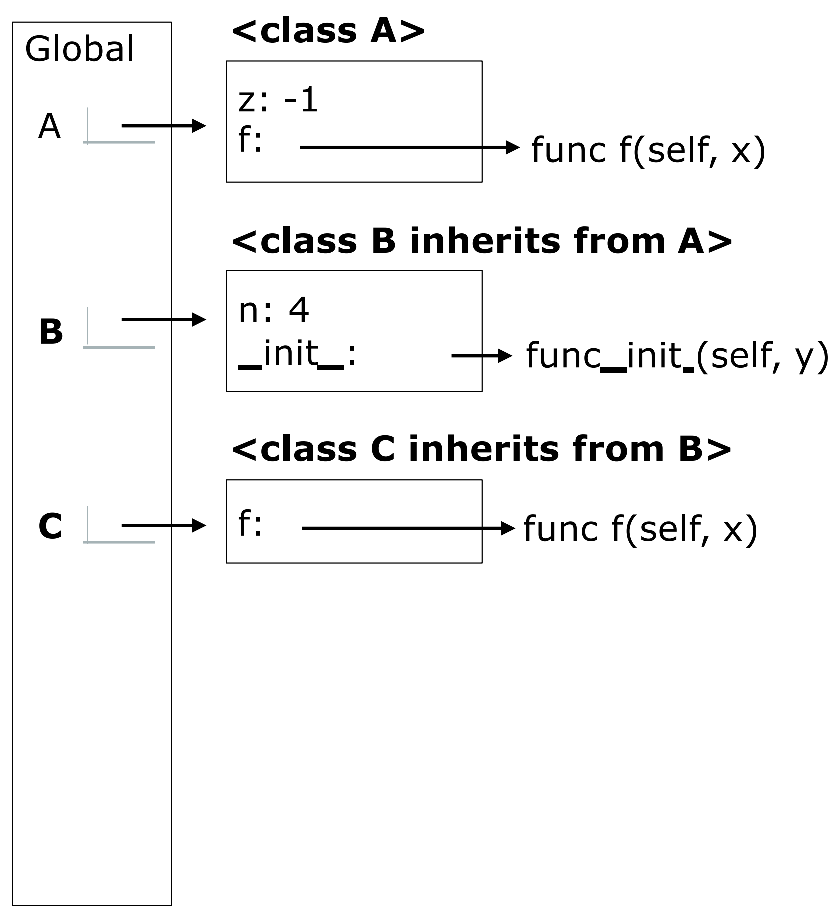
Inheritance and Attribute Lookup

```
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



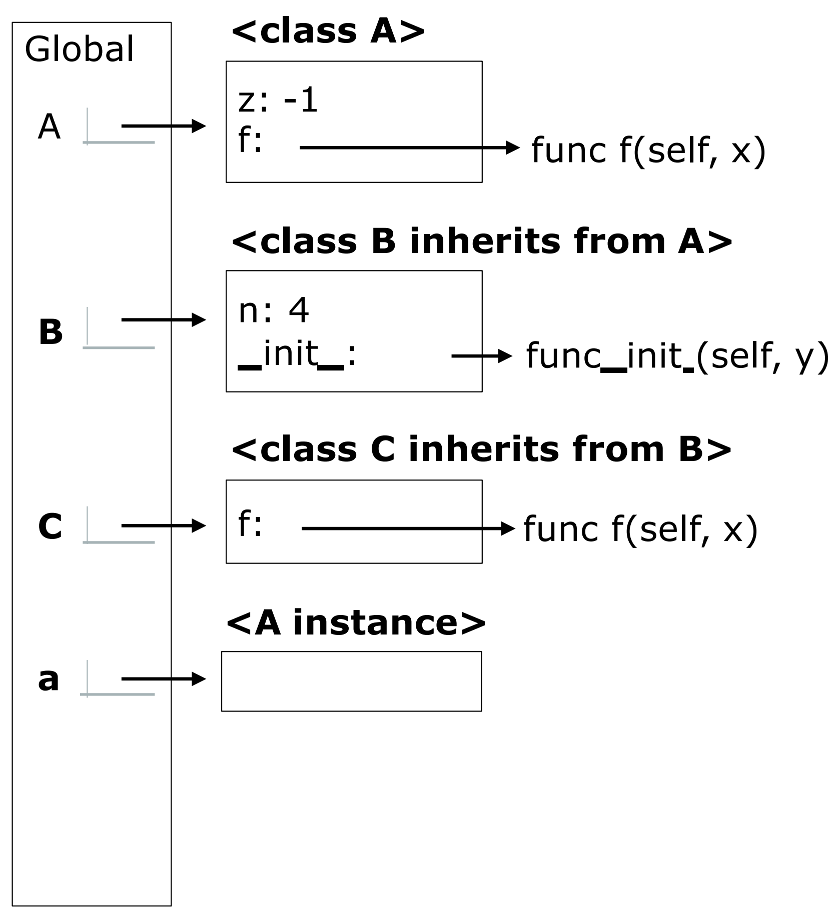
Inheritance and Attribute Lookup

```
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



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
```

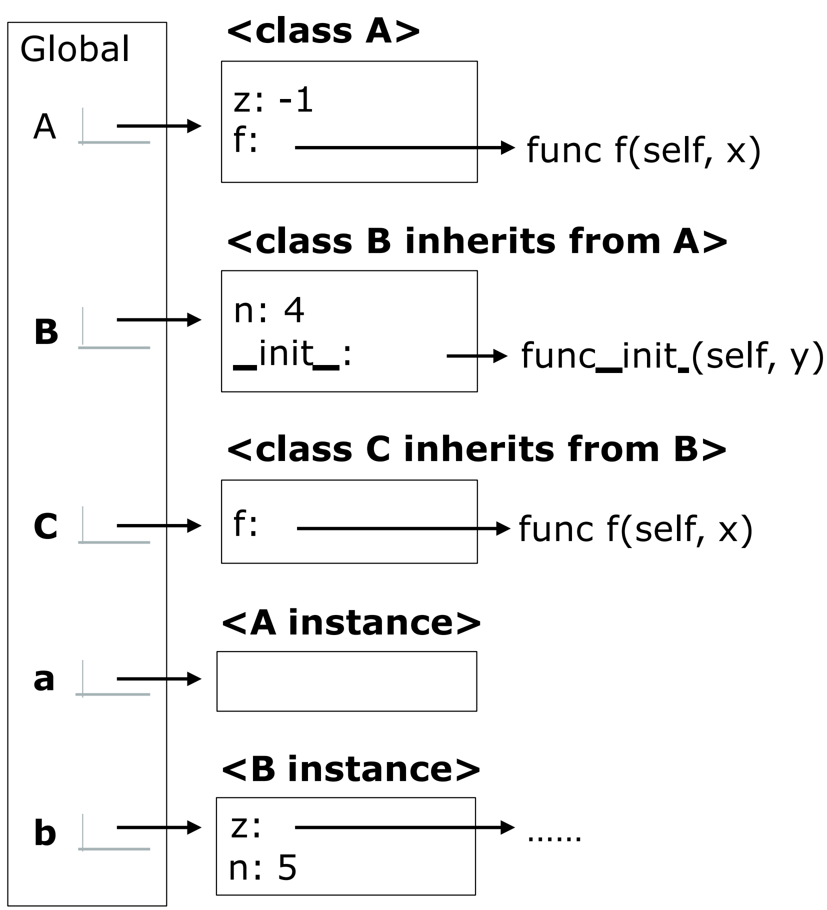
```
>>> C(2).n
```

```
>>> 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
```

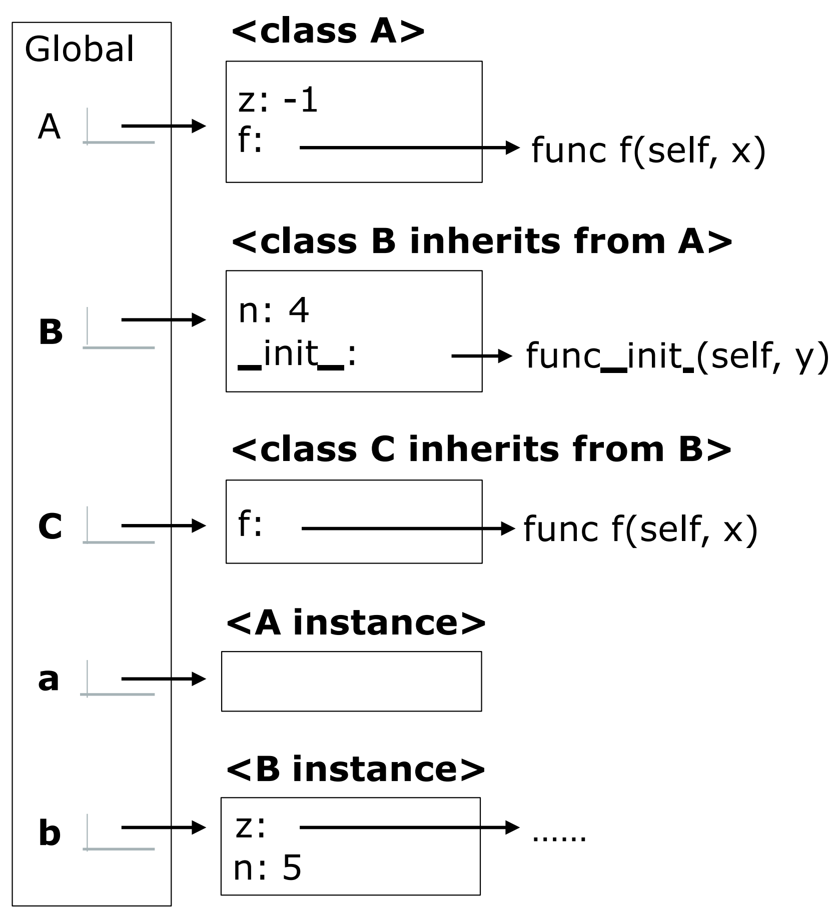
4

```
>>> 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



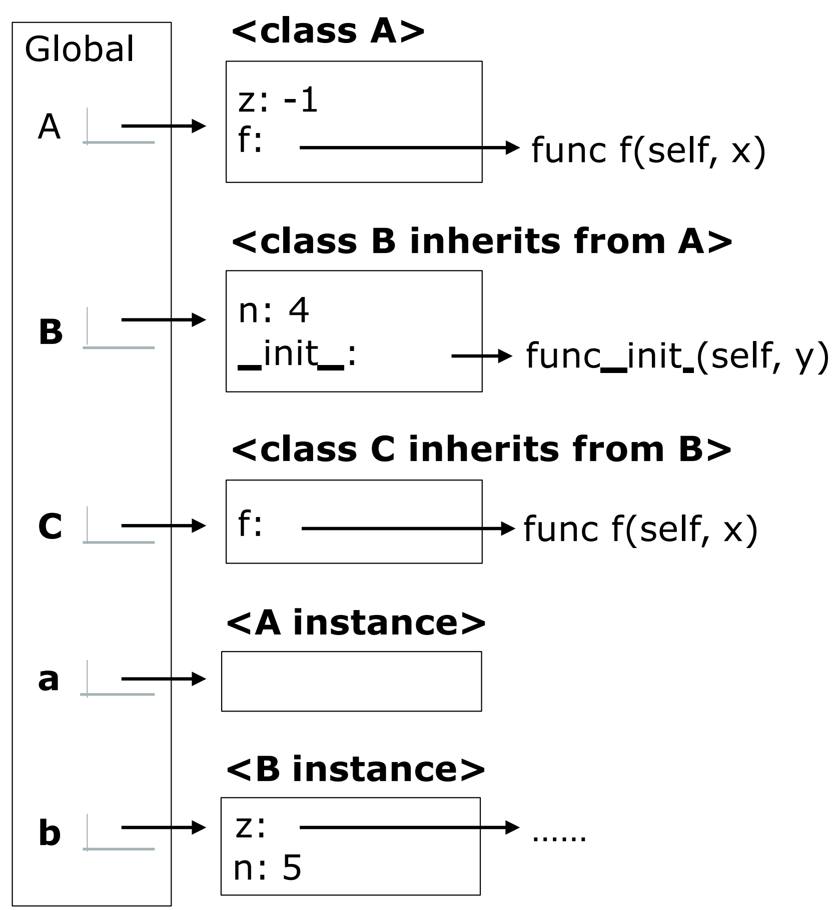
Inheritance and Attribute Lookup

```
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



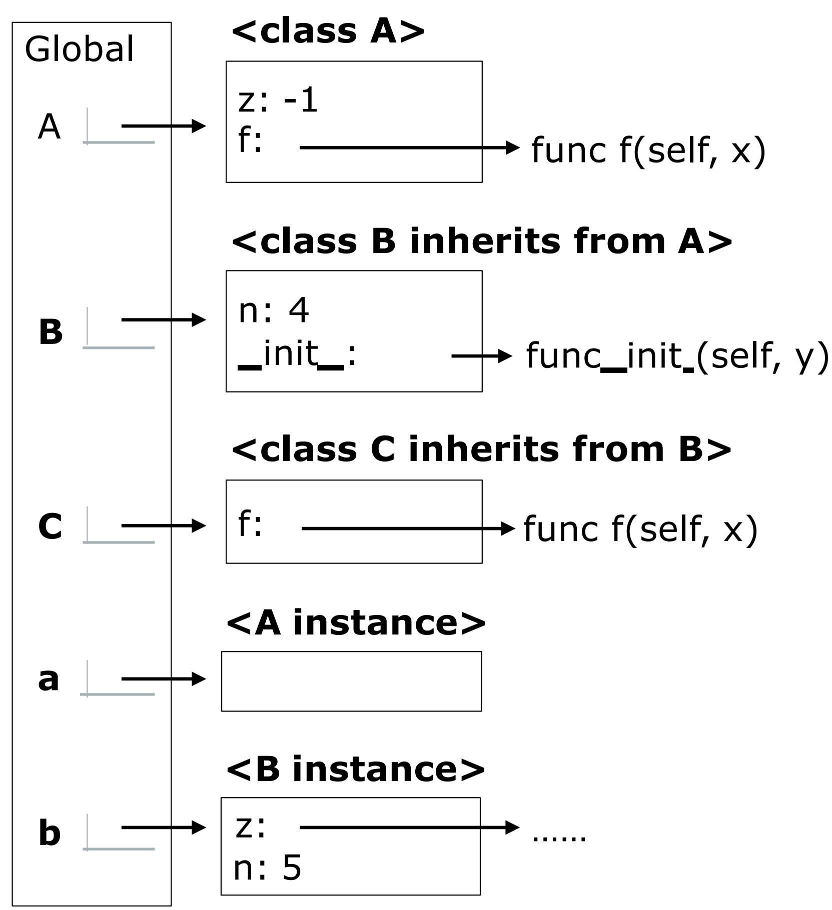
Inheritance and Attribute Lookup

```python
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



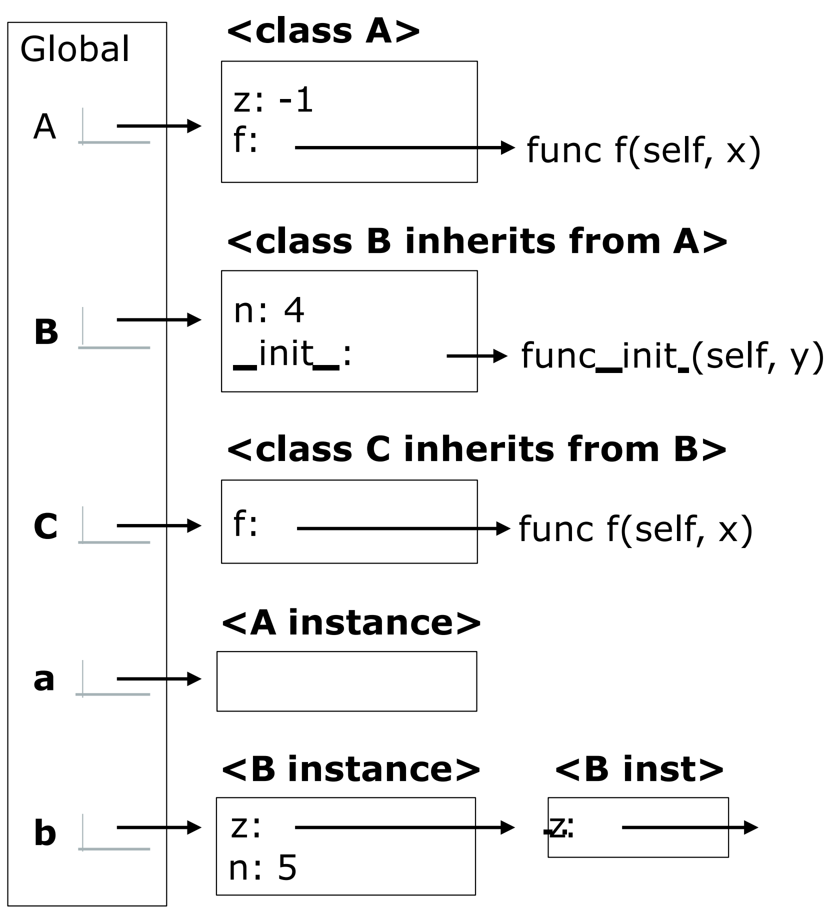
Inheritance and Attribute Lookup

```
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



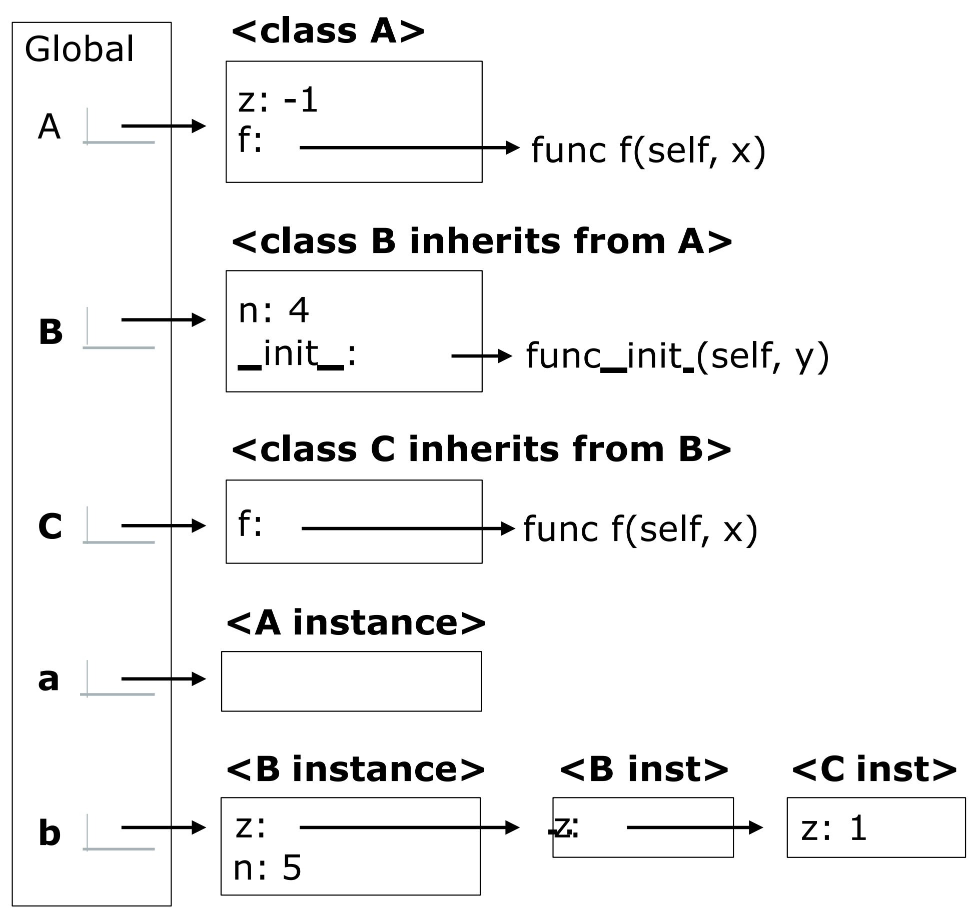
Inheritance and Attribute Lookup

```
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

- Rules of attribute assignment
- Rules of inheritance
- Rules of attribute lookup on classes
- Difference between inheritance and composition