Inheritance

Announcements

Attributes

Methods and Functions

Methods and Functions

```
Python distinguishes between:
```

Methods and Functions

Python distinguishes between:

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

Methods and Functions

Python distinguishes between:

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

- *Bound methods*, which couple together a function and the object on which that method will be invoked

Methods and Functions

Python distinguishes between:

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

- *Bound methods*, which couple together a function and the object on which that method will be invoked

        Object + Function = Bound Method

Methods and Functions

Python distinguishes between:

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

- *Bound methods*, which couple together a function and the object on which that method will be invoked

```
        Object  +  Function  =  Bound Method


        >>> type(Account.deposit)
```

Methods and Functions

Python distinguishes between:

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

- *Bound methods*, which couple together a function and the object on which that method will be invoked

```
    Object + Function = Bound Method


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

Methods and Functions

Python distinguishes between:

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

- *Bound methods*, which couple together a function and the object on which that method will be invoked

```
        Object  +  Function  =  Bound Method


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

Methods and Functions

Python distinguishes between:

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

- *Bound methods*, which couple together a function and the object on which that method will be invoked

```
        Object  +  Function  =  Bound Method


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

Methods and Functions

Python distinguishes between:

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

- *Bound methods*, which couple together a function and the object on which that method will be invoked

```
        Object  +  Function  =  Bound Method


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


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

Methods and Functions

Python distinguishes between:

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

- *Bound methods*, which couple together a function and the object on which that method will be invoked

```
        Object  +  Function  =  Bound Method


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


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

Methods and Functions

Python distinguishes between:

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

• *Bound methods*, which couple together a function and the object on which that method will be invoked

```
Object + Function = Bound Method

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

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

Function: all arguments within parentheses

Methods and Functions

Python distinguishes between:

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

- *Bound methods*, which couple together a function and the object on which that method will be invoked

```
Object  +  Function  =  Bound Method

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

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

Function: all arguments within parentheses

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

Terminology: Attributes, Functions, and Methods

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs
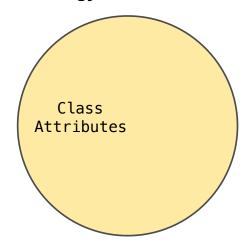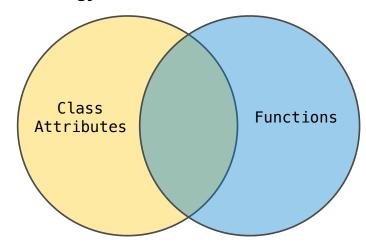
Classes are objects too, so they have attributes

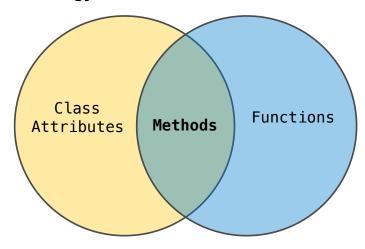# Terminology: Attributes, Functions, and Methods

```
All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

Instance attribute: attribute of an instance
```

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: 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:

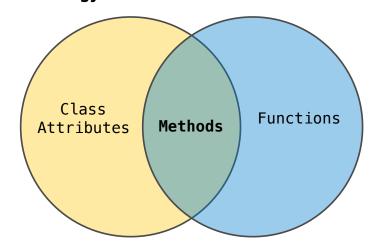# Terminology: Attributes, Functions, and Methods

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:

Class
Attributes

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:

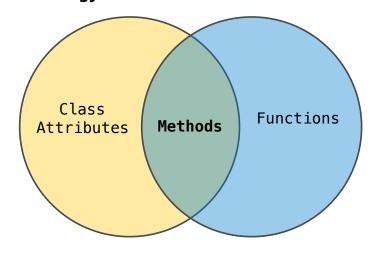# Terminology: Attributes, Functions, and Methods

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:

Class
Attributes   **Methods**   Functions

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:

Venn diagram with overlapping circles: "Class Attributes" (yellow), "Methods" (overlap), "Functions" (blue)
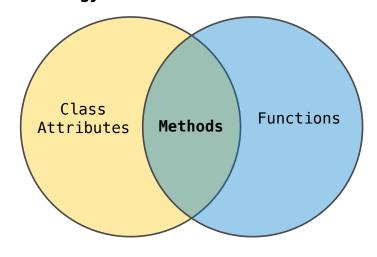
Terminology: Attributes, Functions, and Methods

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:

Class Attributes | Methods | Functions

Python object system:

Functions are objects

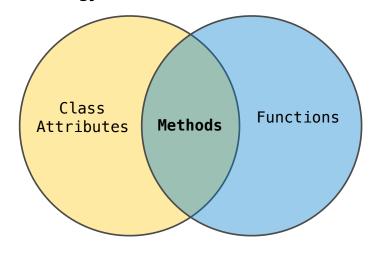# Terminology: Attributes, Functions, and Methods

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:

Class Attributes | Methods | Functions

Python object system:

Functions are objects

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

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:

Class Attributes   **Methods**   Functions

Python object system:

Functions are objects

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

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

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 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>`

Class Attributes   **Methods**   Functions

Looking Up Attributes by Name

`<expression> . <name>`

Looking Up Attributes by Name

<expression> . <name>

To evaluate a dot expression:

Looking Up Attributes by Name

<expression> . <name>

To evaluate a dot expression:

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

Looking Up Attributes by Name

<expression> . <name>

To evaluate a dot expression:

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

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

Looking Up Attributes by Name

<expression> . <name>

To evaluate a dot expression:

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

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

3. If not, <name> is looked up in the class, which yields a class attribute value

Looking Up Attributes by Name

<expression> . <name>

To evaluate a dot expression:

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

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

3. If not, <name> is looked up in the class, which yields a class attribute value

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

Class Attributes

Class Attributes

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

Class Attributes

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

```python
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

```python
class Account:

    interest = 0.02   # A class attribute

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

    # Additional methods would be defined here

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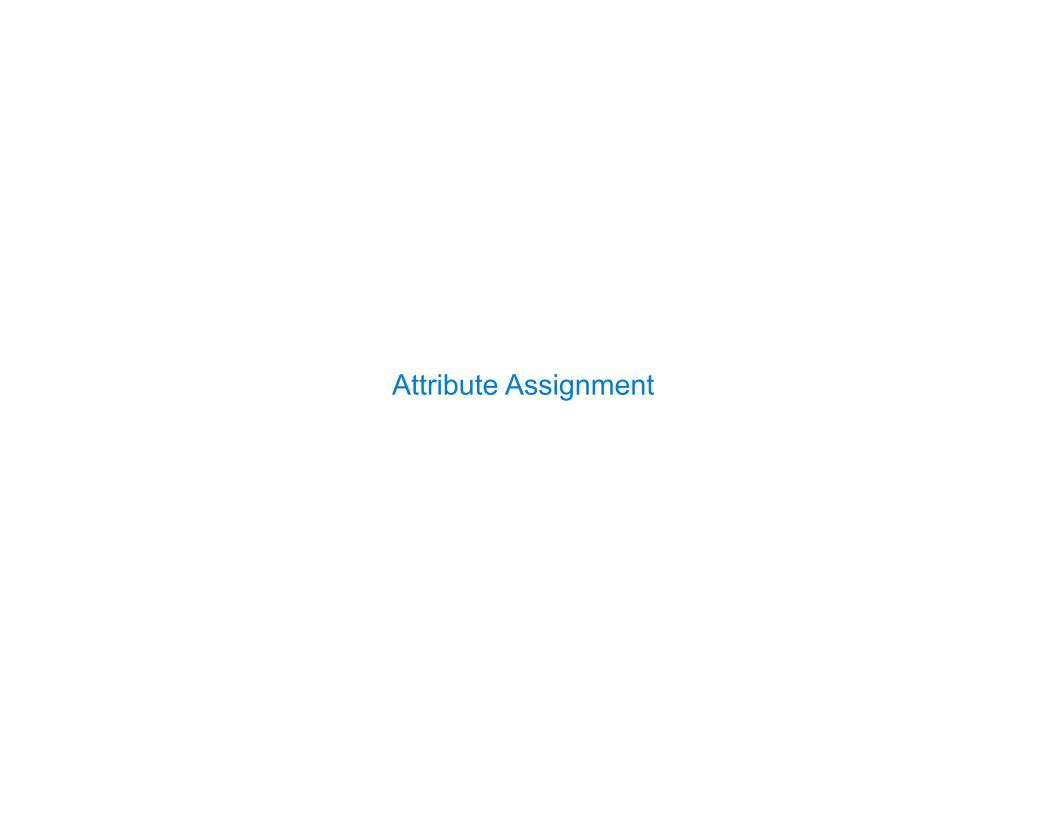
Class Attributes

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

```python
class Account:

    interest = 0.02   # A class attribute

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

    # Additional methods would be defined here

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

Class Attributes

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

```python
class Account:

    interest = 0.02    # A class attribute

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

    # Additional methods would be defined here


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

Class Attributes

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

```python
class Account:

    interest = 0.02   # A class attribute

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

    # Additional methods would be defined here

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

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

Class Attributes

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

```python
class Account:

    interest = 0.02   # A class attribute

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

    # Additional methods would be defined here

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

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

Attribute Assignment

Assignment to Attributes

Assignment to Attributes

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

Assignment to Attributes

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

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

Assignment to Attributes

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

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

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

Assignment to Attributes

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

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

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

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

tom_account = Account('Tom')
```
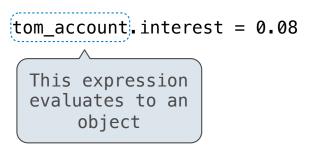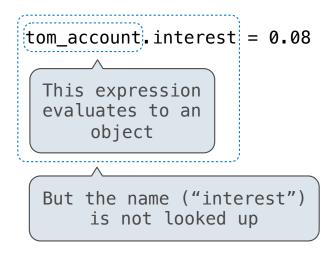
Assignment to Attributes

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

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

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

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

tom_account = Account('Tom')
```

```python
tom_account.interest = 0.08
```
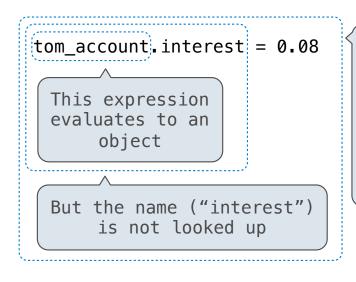
Assignment to Attributes

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

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

tom_account = Account('Tom')
```

tom_account.interest = 0.08

This expression evaluates to an object
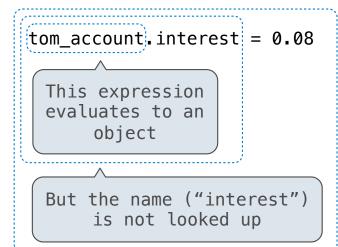
Assignment to Attributes

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

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

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

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

tom_account = Account('Tom')
```

tom_account.interest = 0.08

This expression evaluates to an object

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

Assignment to Attributes

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

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

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

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

tom_account = Account('Tom')
```

tom_account.interest = 0.08

This expression evaluates to an object

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

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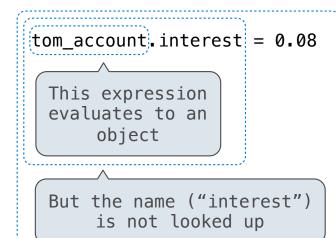
Assignment to Attributes

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

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

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

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

tom_account = Account('Tom')
```

Instance
Attribute
Assignment

:   tom_account.interest = 0.08

This expression evaluates to an object

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

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

Assignment to Attributes

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

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

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

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

tom_account = Account('Tom')
```

Instance
Attribute   :   tom_account.interest = 0.08
Assignment

This expression evaluates to an object

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

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

Class
Attribute   :   Account.interest = 0.04
Assignment

Attribute Assignment Statements

Account class attributes

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

Attribute Assignment Statements

Account class
attributes

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

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

Attribute Assignment Statements

Account class attributes

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

Instance attributes of jim_account

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

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

Attribute Assignment Statements

Account class
attributes

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

Instance
attributes of
jim_account

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

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

Attribute Assignment Statements

Attribute Assignment Statements

Account class attributes

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

Instance attributes of jim_account

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

Instance attributes of tom_account

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

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

Attribute Assignment Statements

Account class attributes

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

Instance attributes of jim_account

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

Instance attributes of tom_account

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

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

Attribute Assignment Statements

Account class attributes

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

Instance attributes of jim_account

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

Instance attributes of tom_account

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

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

Attribute Assignment Statements

Account class attributes

```
interest: 0̶.̶0̶2̶  0.04
(withdraw, deposit, __init__)
```

Instance attributes of jim_account

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

Instance attributes of tom_account

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

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

Attribute Assignment Statements

Account class attributes

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

Instance attributes of jim_account

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

Instance attributes of tom_account

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

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

Attribute Assignment Statements

Account class attributes ▷

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

Instance attributes of jim_account ▷

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

Instance attributes of tom_account ▷

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

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

Attribute Assignment Statements

Account class attributes →

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

Instance attributes of jim_account →

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

Instance attributes of tom_account →

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

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

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

Attribute Assignment Statements

Account class
attributes

```
interest: 0̶.̶0̶2̶  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     interest: 0̶.̶0̶2̶  0.04
attributes        (withdraw, deposit, __init__)
```

```
Instance           balance:  0
attributes of      holder:   'Jim'
jim_account        interest: 0.08
```

```
Instance           balance:  0
attributes of      holder:   'Tom'
tom_account
```

```
>>> 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: 0̶.̶0̶2̶  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: 0̶.̶0̶2̶  0.04
(withdraw, deposit, __init__)
```

Instance attributes of jim_account ▷
```
balance:  0
holder:    'Jim'
interest: 0.08
```

Instance attributes of tom_account ▷
```
balance:  0
holder:    'Tom'
```

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

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

Attribute Assignment Statements

Account class attributes →
```
interest: 0̶.̶0̶2̶  0̶.̶0̶4̶  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: 0̶.̶0̶2̶  0̶.̶0̶4̶  0.05
(withdraw, deposit, __init__)
```

Instance attributes of jim_account →
```
balance:  0
holder:   'Jim'
interest: 0.08
```

Instance attributes of tom_account →
```
balance:  0
holder:   'Tom'
```

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

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

Attribute Assignment Statements

Account class attributes

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

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

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

Inheritance

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

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>
```

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

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')
```

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
```

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

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):
```

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."""
```

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
```

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
```

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):
```

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)
```

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)
```

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)

                                    or

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

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

Looking Up Attribute Names on Classes

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

Looking Up Attribute Names on Classes

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

To look up a name in a class:

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.

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.
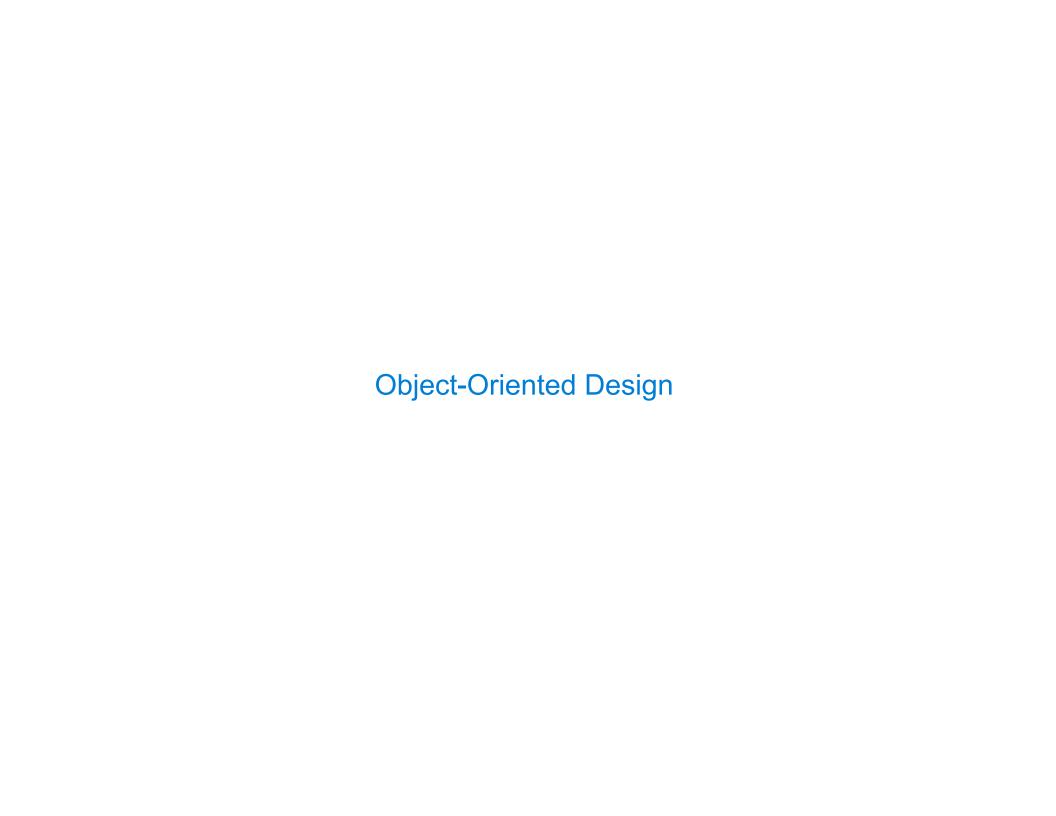
Looking Up Attribute Names on Classes

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__
```

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)

Object-Oriented Design

Designing for Inheritance

Designing for Inheritance

Don't repeat yourself; use existing implementations

Designing for Inheritance

Don't repeat yourself; use existing implementations

```python
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

```python
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

```python
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

```python
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

```python
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

Inheritance and Composition

Object-oriented programming shines when we adopt the metaphor

Inheritance and Composition

Object-oriented programming shines when we adopt the metaphor

Inheritance is best for representing is-a relationships

Inheritance and Composition

Object-oriented programming shines when we adopt the metaphor

Inheritance is best for representing is-a relationships

- E.g., a checking account is a specific type of account

Inheritance and Composition

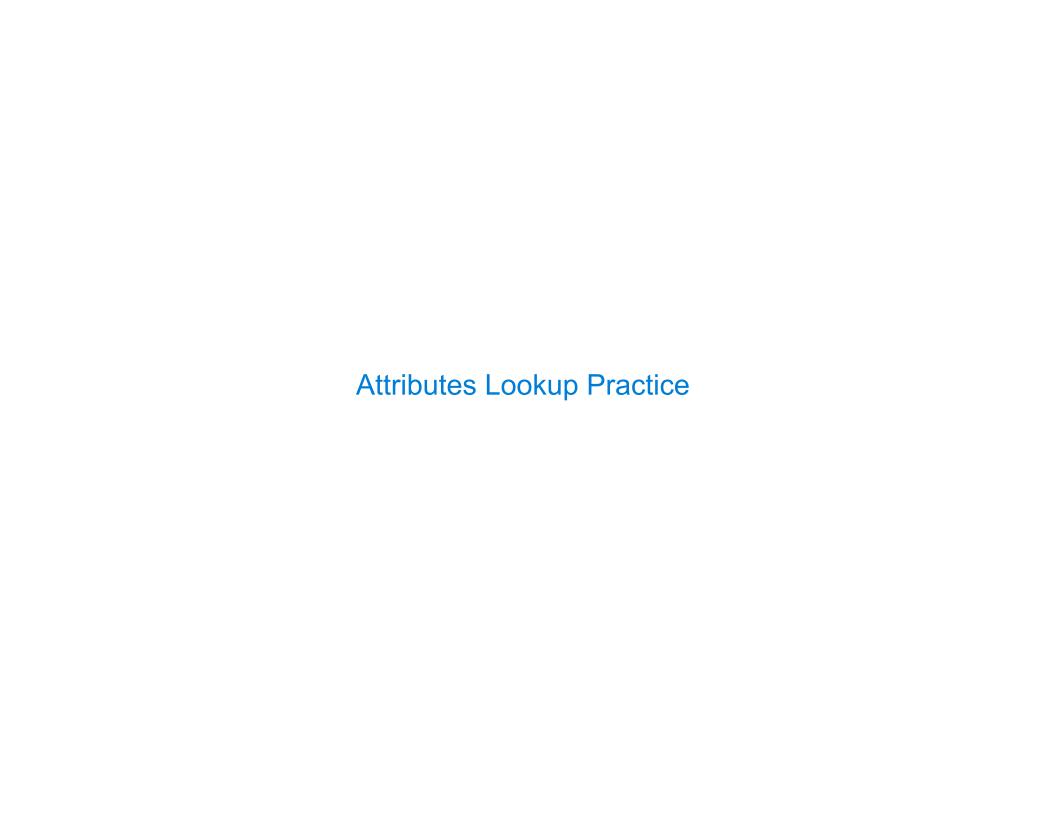Object-oriented programming shines when we adopt the metaphor

Inheritance is best for representing is-a relationships

- E.g., a checking account is a specific type of account

- So, CheckingAccount inherits from Account

Inheritance and Composition

Object-oriented programming shines when we adopt the metaphor

Inheritance is best for representing is-a relationships

- E.g., a checking account is a specific type of account

- So, CheckingAccount inherits from Account

Composition is best for representing has-a relationships

Inheritance and Composition

Object-oriented programming shines when we adopt the metaphor

Inheritance is best for representing is-a relationships

• E.g., a checking account 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

Inheritance and Composition

Object-oriented programming shines when we adopt the metaphor

Inheritance is best for representing is-a relationships

- E.g., a checking account 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

Inheritance and Composition

Object-oriented programming shines when we adopt the metaphor

Inheritance is best for representing is-a relationships

• E.g., a checking account 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

(Demo)

Attributes Lookup Practice
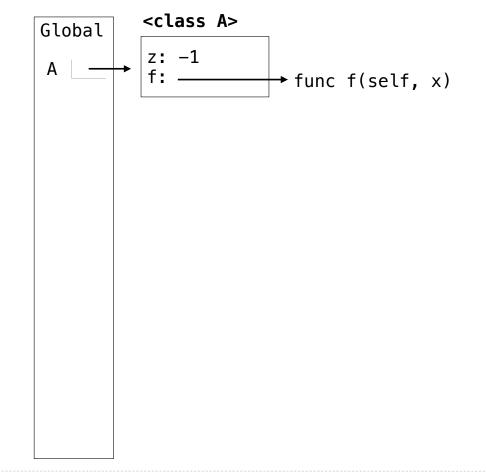
Inheritance and Attribute Lookup

```
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

```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


>>> 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
```

```
<class A>
```

```
┌─────────┐
│ z: -1   │
│ f: ──────────→ func f(self, x)
└─────────┘
```

19

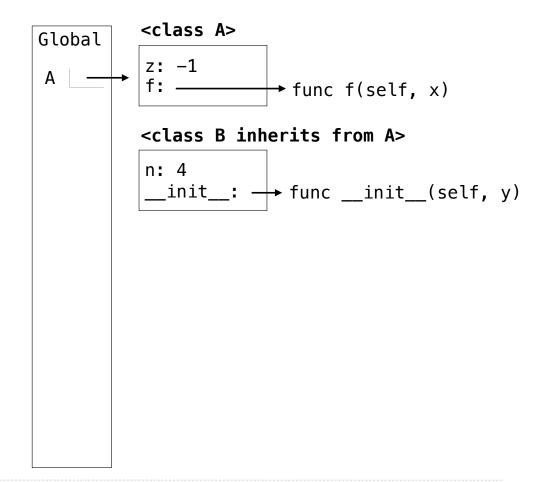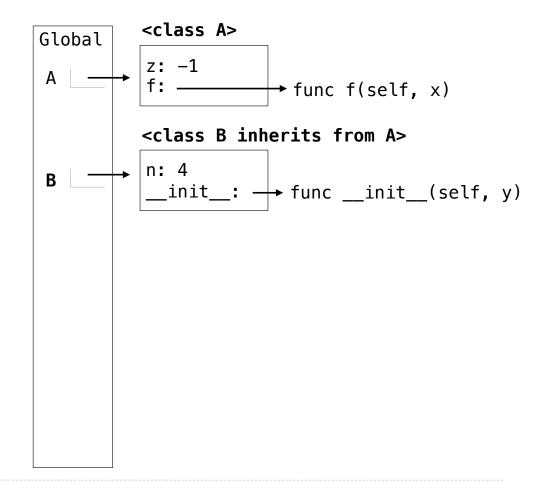# Inheritance and Attribute Lookup

```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


>>> 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
```

<class A>

```
z: -1
f: ──────────→ func f(self, x)
```

19

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
```

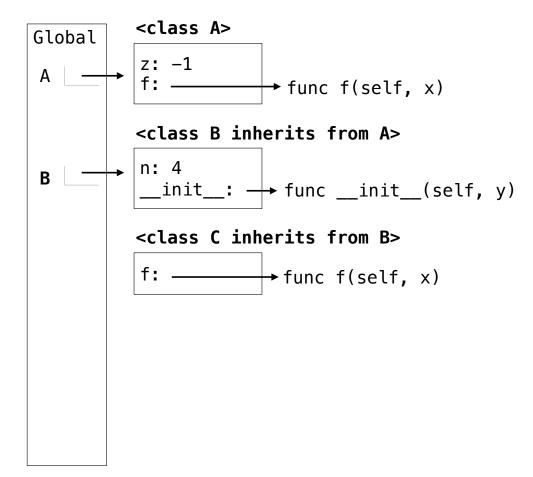# Inheritance and Attribute Lookup

```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


>>> 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

```
┌──────────┐    <class A>
│ Global   │    ┌────────────────┐
│          │    │ z: -1          │
│ A  ──────┼──▶ │ f: ───────────┼──▶ func f(self, x)
│          │    └────────────────┘
│          │    <class B inherits from A>
│          │    ┌────────────────┐
│          │    │ n: 4           │
│          │    │ __init__: ─────┼──▶ func __init__(self, y)
│          │    └────────────────┘
│          │
│          │
│          │
└──────────┘
```

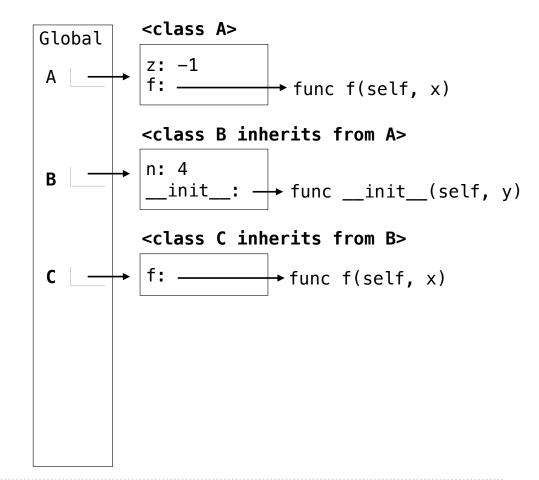# Inheritance and Attribute Lookup

```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


>>> 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

```
            <class A>
Global     ┌─────────────────┐
           │ z: -1           │
A   ──────▶│ f: ───────────────▶ func f(self, x)
           └─────────────────┘

            <class B inherits from A>
           ┌─────────────────┐
B   ──────▶│ n: 4            │
           │ __init__: ───────▶ func __init__(self, y)
           └─────────────────┘
```

19

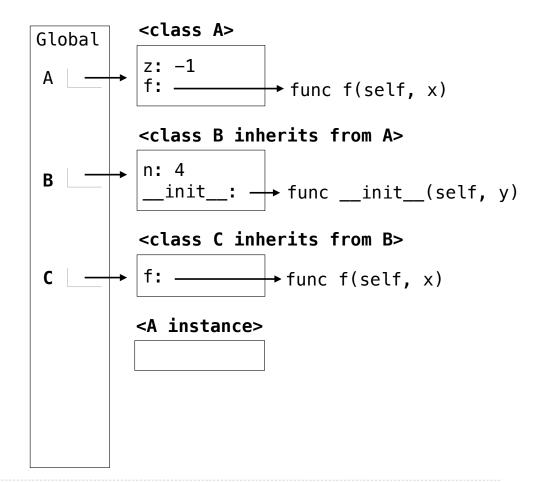# Inheritance and Attribute Lookup

```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


>>> 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
```

<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)
```

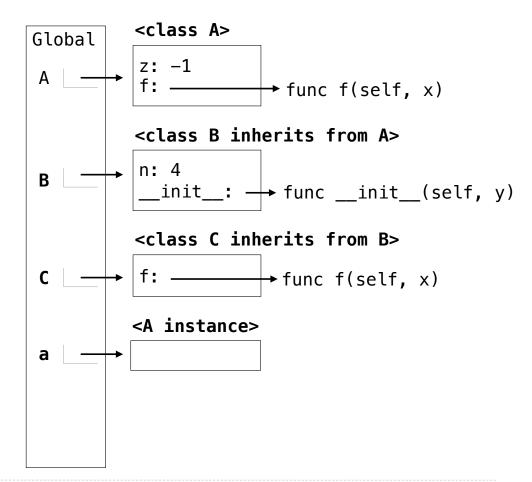# Inheritance and Attribute Lookup

```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


>>> 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
```

\<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)
```

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
```

<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>

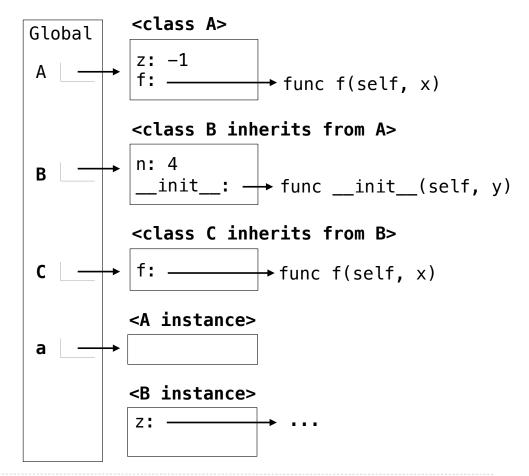# Inheritance and Attribute Lookup

```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


>>> 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
```

```
┌─────────┐   <class A>
│ Global  │   ┌─────────────────┐
│         │   │ z: -1           │
│ A  ─────┼──→│ f: ─────────────┼──→ func f(self, x)
│         │   └─────────────────┘
│         │   <class B inherits from A>
│         │   ┌─────────────────┐
│ B  ─────┼──→│ n: 4            │
│         │   │ __init__: ──────┼──→ func __init__(self, y)
│         │   └─────────────────┘
│         │   <class C inherits from B>
│         │   ┌─────────────────┐
│ C  ─────┼──→│ f: ─────────────┼──→ func f(self, x)
│         │   └─────────────────┘
│         │   <A instance>
│         │   ┌─────────────────┐
│ a  ─────┼──→│                 │
│         │   └─────────────────┘
└─────────┘
```

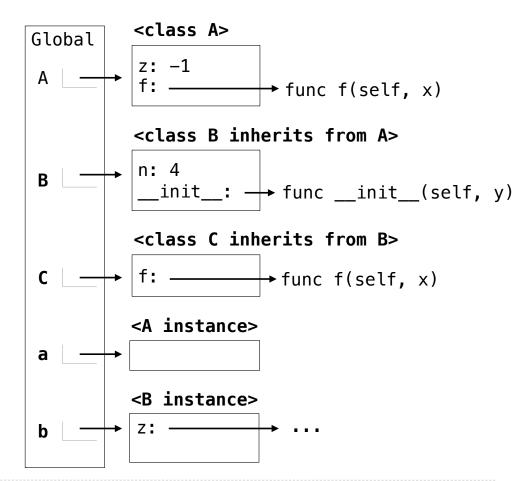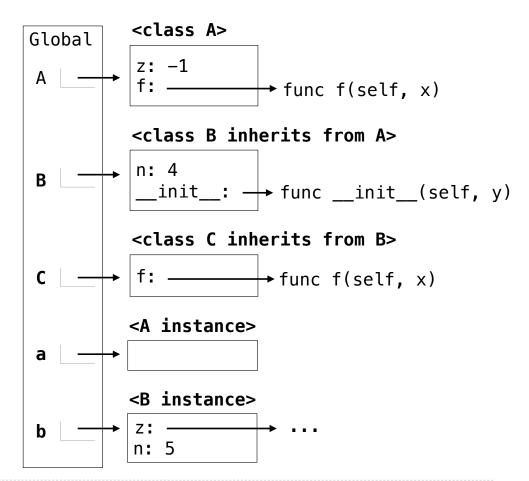# Inheritance and Attribute Lookup

```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

>>> 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        <class A>

A  ────────►  ┌─────────────┐
              │ z: -1       │
              │ f: ─────────┼──► func f(self, x)
              └─────────────┘

              <class B inherits from A>

B  ────────►  ┌─────────────┐
              │ n: 4        │
              │ __init__: ──┼──► func __init__(self, y)
              └─────────────┘

              <class C inherits from B>

C  ────────►  ┌─────────────┐
              │ f: ─────────┼──► func f(self, x)
              └─────────────┘

              <A instance>

a  ────────►  ┌─────────────┐
              │             │
              └─────────────┘

              <B instance>

              ┌─────────────┐
              │ z: ─────────┼──► ...
              └─────────────┘
```
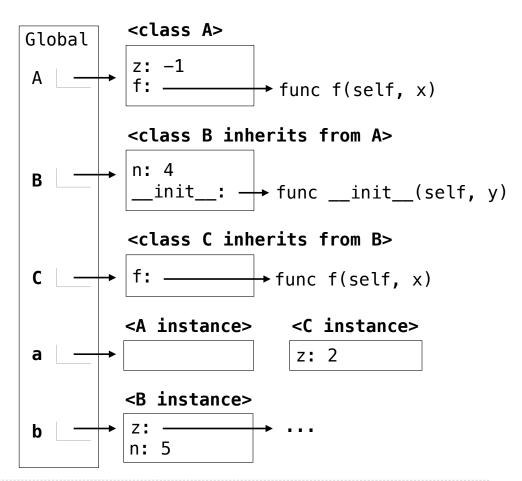
Inheritance and Attribute Lookup

```
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 ────────► <class A>
            ┌─────────────┐
            │ z: -1       │
            │ f: ─────────┼──► func f(self, x)
            └─────────────┘

            <class B inherits from A>
B ────────► ┌─────────────┐
            │ n: 4        │
            │ __init__: ──┼──► func __init__(self, y)
            └─────────────┘

            <class C inherits from B>
C ────────► ┌─────────────┐
            │ f: ─────────┼──► func f(self, x)
            └─────────────┘

            <A instance>
a ────────► ┌─────────────┐
            └─────────────┘

            <B instance>
b ────────► ┌─────────────┐
            │ z: ─────────┼──► ...
            └─────────────┘
```

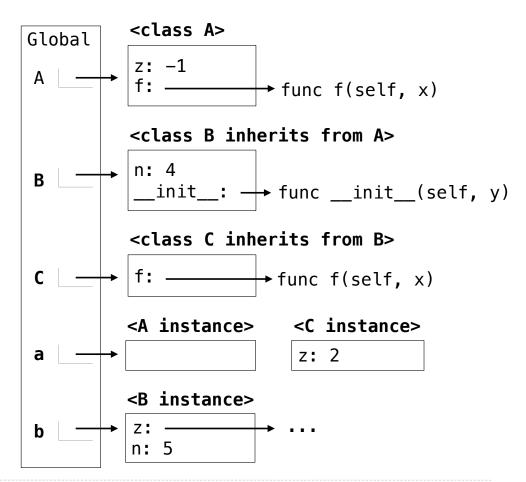19

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
```

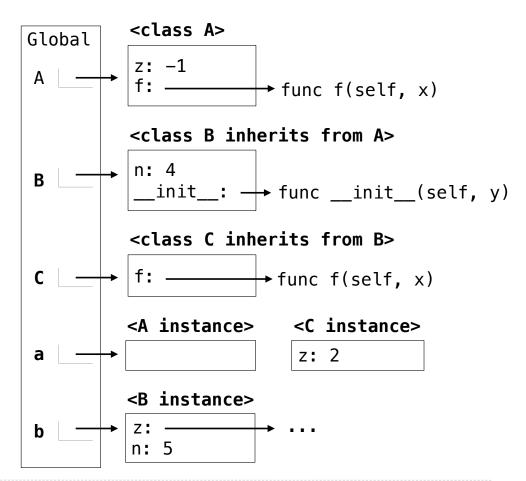# Inheritance and Attribute Lookup

```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


>>> 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 ──────→  <class A>
           ┌──────────────┐
           │ z: -1        │
           │ f: ──────────┼──→ func f(self, x)
           └──────────────┘

           <class B inherits from A>
           ┌──────────────┐
B ──────→  │ n: 4         │
           │ __init__: ───┼──→ func __init__(self, y)
           └──────────────┘

           <class C inherits from B>
           ┌──────────────┐
C ──────→  │ f: ──────────┼──→ func f(self, x)
           └──────────────┘

           <A instance>        <C instance>
           ┌──────────────┐    ┌──────────┐
a ──────→  │              │    │ z: 2     │
           └──────────────┘    └──────────┘

           <B instance>
           ┌──────────────┐
b ──────→  │ z: ──────────┼──→ ...
           │ n: 5         │
           └──────────────┘
```

19

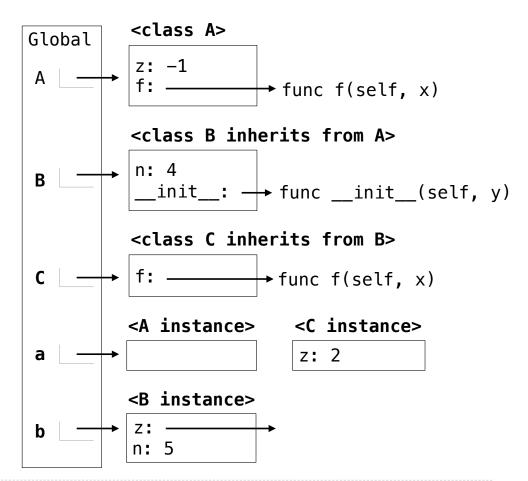# Inheritance and Attribute Lookup

```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


>>> 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	<class A>
	z: -1
A	f: → func f(self, x)

	<class B inherits from A>
	n: 4
B	__init__: → func __init__(self, y)

	<class C inherits from B>
C	f: → func f(self, x)

<A instance>	<C instance>
a	z: 2

<B instance>
z: → ...
b n: 5

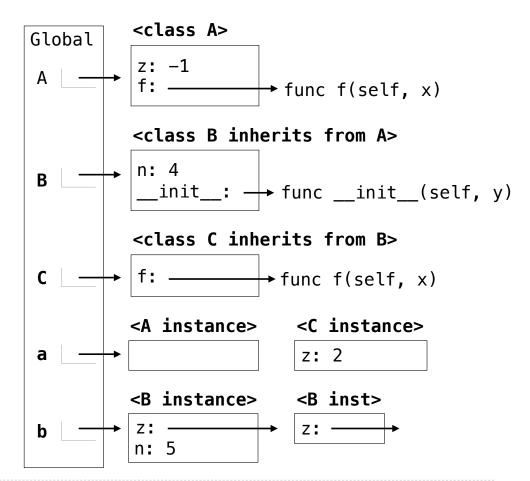# Inheritance and Attribute Lookup

```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
```

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

19

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
```

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>

<C instance>

| z: 2 |

<B instance>

| z: ──→ |
| n: 5 |

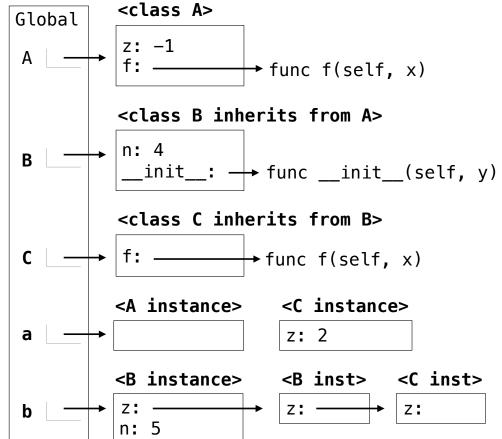# Inheritance and Attribute Lookup

```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
```

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

```
Global          <class A>
                ┌──────────────┐
A  ────────────→│ z: -1        │
                │ f: ─────────────→ func f(self, x)
                └──────────────┘

                <class B inherits from A>
                ┌──────────────┐
B  ────────────→│ n: 4         │
                │ __init__: ──────→ func __init__(self, y)
                └──────────────┘

                <class C inherits from B>
                ┌──────────────┐
C  ────────────→│ f: ─────────────→ func f(self, x)
                └──────────────┘

   <A instance>          <C instance>
   ┌──────────────┐      ┌──────────────┐
a ─│              │      │ z: 2         │
   └──────────────┘      └──────────────┘

   <B instance>          <B inst>
   ┌──────────────┐      ┌──────────────┐
b ─│ z: ─────────────→   │ z: ─────────────→
   │ n: 5         │      └──────────────┘
   └──────────────┘
```

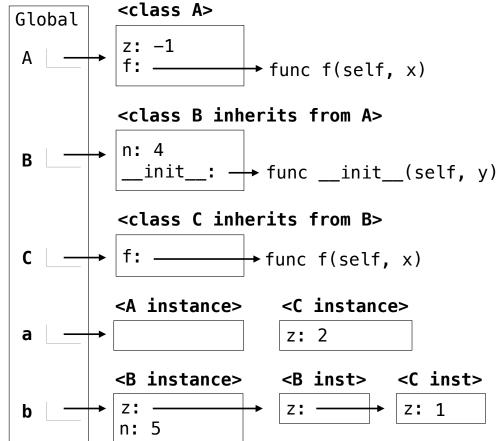# Inheritance and Attribute Lookup

```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
```

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>

<C instance>

z: 2

<B instance>

z: ────
n: 5

<B inst>

z: ────

<C inst>

z:

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
```

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

Global	<class A>
A →	z: -1 f: → func f(self, x)
B →	**<class B inherits from A>** n: 4 __init__: → func __init__(self, y)
C →	**<class C inherits from B>** f: → func f(self, x)
a →	**<A instance>** **<C instance>** z: 2
b →	**<B instance>** z: → n: 5 **<B inst>** z: → **<C inst>** z: 1

19

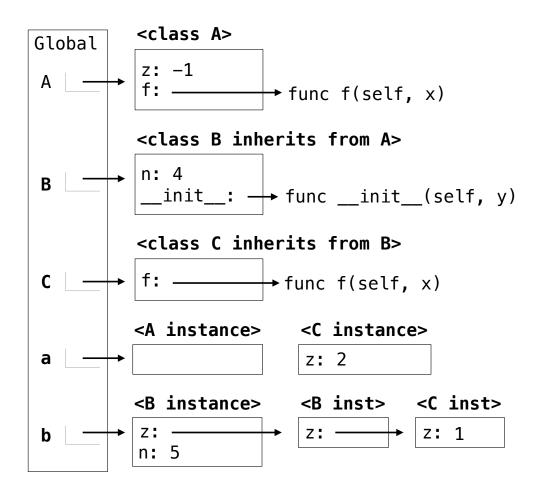# Inheritance and Attribute Lookup

```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
```

```
Global          <class A>
                ┌──────────────┐
A  ──────────►  │ z: -1        │
                │ f: ──────────┼──► func f(self, x)
                └──────────────┘

                <class B inherits from A>
                ┌──────────────┐
B  ──────────►  │ n: 4         │
                │ __init__: ───┼──► func __init__(self, y)
                └──────────────┘

                <class C inherits from B>
                ┌──────────────┐
C  ──────────►  │ f: ──────────┼──► func f(self, x)
                └──────────────┘

                <A instance>        <C instance>
                ┌──────────────┐    ┌──────────┐
a  ──────────►  │              │    │ z: 2     │
                └──────────────┘    └──────────┘

                <B instance>        <B inst>      <C inst>
                ┌──────────────┐    ┌────────┐   ┌────────┐
b  ──────────►  │ z: ──────────┼──► │ z: ────┼──►│ z: 1   │
                │ n: 5         │    └────────┘   └────────┘
                └──────────────┘
```

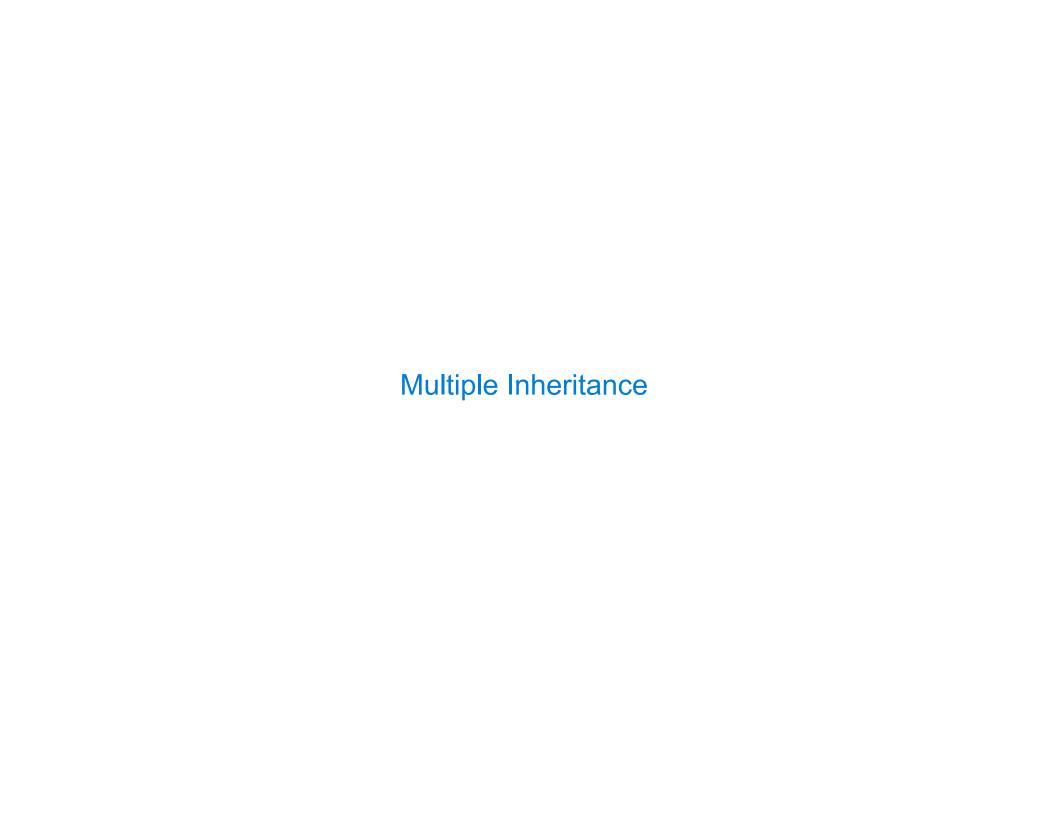# Inheritance and Attribute Lookup

```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

Global	**<class A>**
A →	z: -1 f: → func f(self, x)
	<class B inherits from A>
B →	n: 4 __init__: → func __init__(self, y)
	<class C inherits from B>
C →	f: → func f(self, x)

<A instance> **<C instance>**

a → [   ]   z: 2

<B instance> **<B inst>** **<C inst>**

b → z: →
 n: 5   z: →   z: 1

19

Multiple Inheritance

Multiple Inheritance

Multiple Inheritance

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

Multiple Inheritance

```python
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

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:

Multiple Inheritance

```python
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%

Multiple Inheritance

```python
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

Multiple Inheritance

```python
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

Multiple Inheritance

```python
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

Multiple Inheritance

```python
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

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

Multiple Inheritance

A class may inherit from multiple base classes in Python.

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

Multiple Inheritance

A class may inherit from multiple base classes in Python.

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

```python
>>> such_a_deal = AsSeenOnTVAccount('John')
```

Multiple Inheritance

A class may inherit from multiple base classes in Python.

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

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

Multiple Inheritance

A class may inherit from multiple base classes in Python.

```python
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

A class may inherit from multiple base classes in Python.

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

Instance attribute

```python
>>> such_a_deal = AsSeenOnTVAccount('John')
>>> such_a_deal.balance
1
>>> such_a_deal.deposit(20)
19
```

Multiple Inheritance

A class may inherit from multiple base classes in Python.

```python
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
```
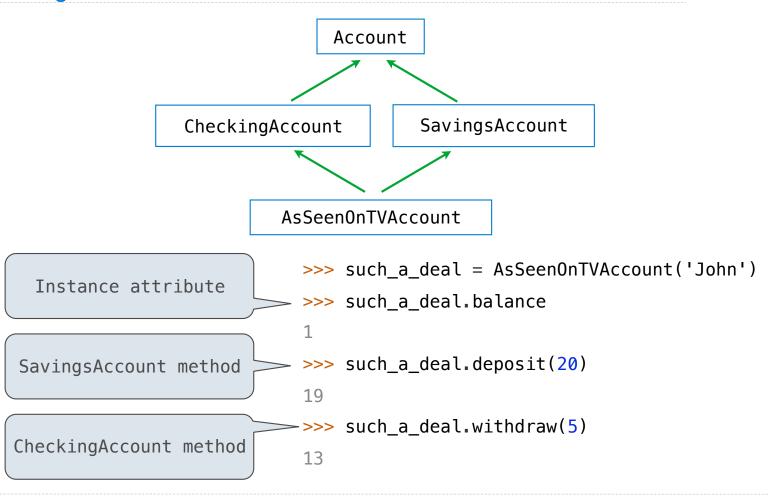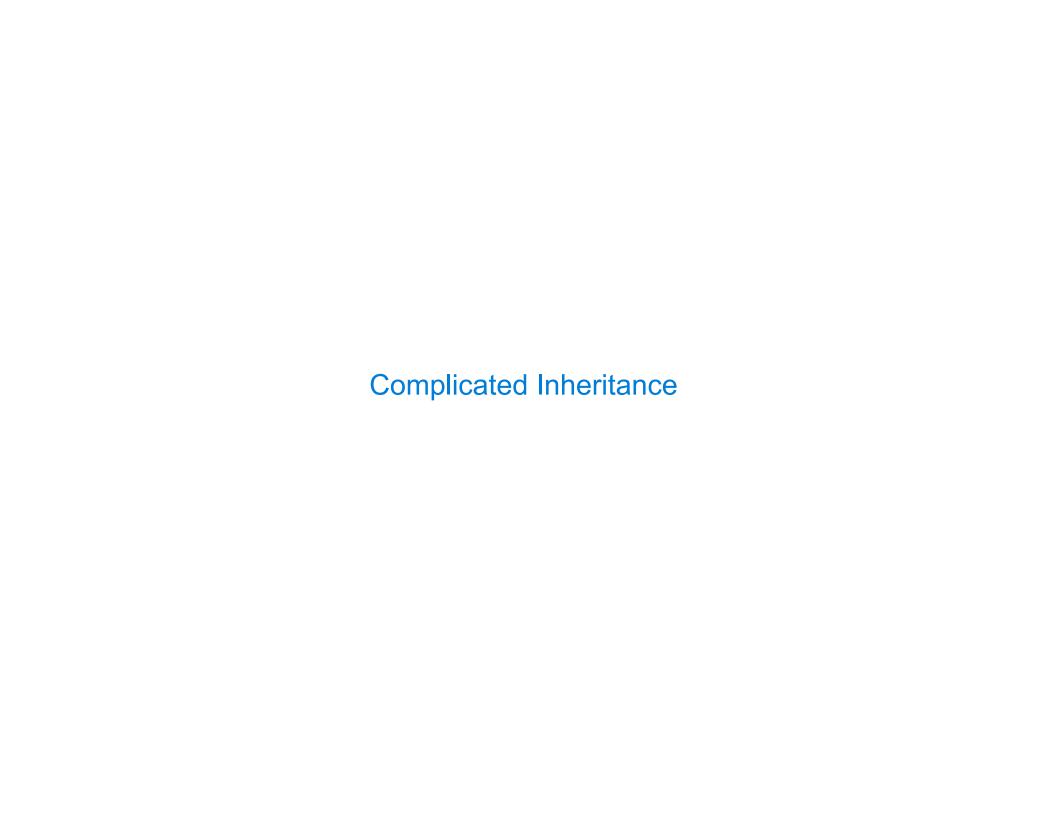
SavingsAccount method

```
>>> such_a_deal.deposit(20)
19
```

Multiple Inheritance
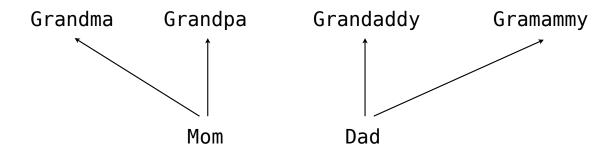
A class may inherit from multiple base classes in Python.

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

Instance attribute

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

SavingsAccount method

```python
>>> such_a_deal.deposit(20)
19
>>> such_a_deal.withdraw(5)
13
```

Multiple Inheritance

A class may inherit from multiple base classes in Python.

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

Instance attribute

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

SavingsAccount method

```python
>>> such_a_deal.deposit(20)
19
```

CheckingAccount method

```python
>>> such_a_deal.withdraw(5)
13
```

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
```

Resolving Ambiguous Class Attribute Names


```
Account
```

```
CheckingAccount          SavingsAccount
```

```
AsSeenOnTVAccount
```

Instance attribute

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

SavingsAccount method

```
>>> such_a_deal.deposit(20)
19
```

CheckingAccount method

```
>>> such_a_deal.withdraw(5)
13
```

23

Complicated Inheritance

Biological Inheritance

Biological Inheritance

Grandma   Grandpa   Grandaddy   Gramammy

Biological Inheritance

Biological Inheritance

Biological Inheritance

Biological Inheritance

Biological Inheritance

Biological Inheritance

Biological Inheritance

Biological Inheritance

Biological Inheritance

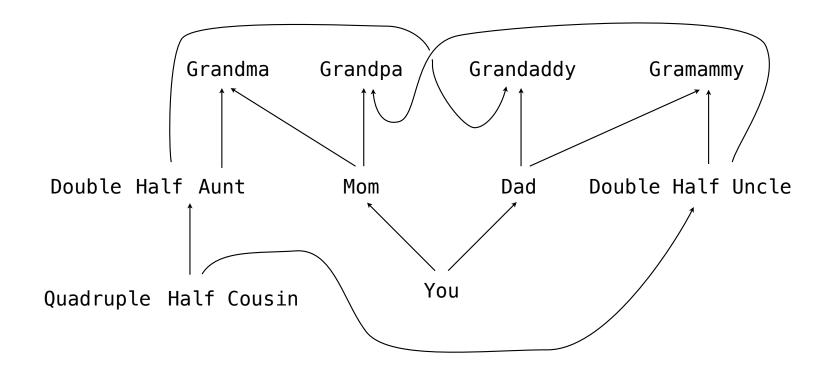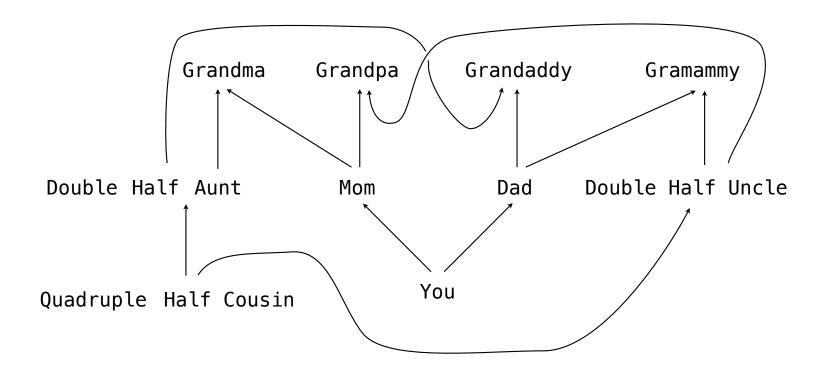# Biological Inheritance

Biological Inheritance

Moral of the story: Inheritance can be complicated, so don't overuse it!