Announcements
Attributes
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.
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: 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: 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

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>
Reminder: Looking Up Attributes by Name

<expression> . <name>
Reminder: Looking Up Attributes by Name

<expression> . <name>

To evaluate a dot expression:
Reminder: 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.
Reminder: 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
 Reminder: Looking Up Attributes by 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
Reminder: 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
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

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

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

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

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

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

tom_account = Account('Tom')
tom_account.interest = 0.08
Assignment to Attributes

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

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

```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

```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
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

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

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')
```

Instance : `tom_account.interest = 0.08`

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

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')
```

Instance : `tom_account.interest = 0.08`

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

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

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

Class : `Account.interest = 0.04`

This expression evaluates to an object
Attribute Assignment Statements

Account class attributes

```python
interest: 0.02
(withdraw, deposit, __init__)
```
Attribute Assignment Statements

```python
>>> 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

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

Account class attributes

- interest: 0.02
  (withdraw, deposit, __init__)  

Instance attributes of jim_account

- balance: 0
- holder: 'Jim'

Instance attributes of tom_account

- balance: 0
- holder: 'Tom'
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'

```python
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
```
Attribute Assignment Statements

Account class attributes

<table>
<thead>
<tr>
<th>balance: 0</th>
<th>holder: 'Jim'</th>
</tr>
</thead>
</table>

| balance: 0 | holder: 'Tom' |

Instance attributes of jim_account

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

Instance attributes of tom_account

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

```python
>>> 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

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

Instance attributes of jim_account

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

Instance attributes of tom_account

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

```python
>>> 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**

```python
>>> 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'

```python
>>> 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

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

Account class attributes

| interest: 0.02 | 0.04 |
|----------------|
| (withdraw, deposit, __init__) |

Instance attributes of `jim_account`

- balance: 0
- holder: 'Jim'

Instance attributes of `tom_account`

- balance: 0
- holder: 'Tom'
Attribute Assignment Statements

```python
>>> 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
```

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

Account class attributes:
- `interest`: 0.02, 0.04 (withdraw, deposit, __init__)

Instance attributes of `jim_account`:
- balance: 0
- holder: 'Jim'

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

Attribute Assignment Statements

Account class attributes

<table>
<thead>
<tr>
<th>interest</th>
<th>0.02 0.04</th>
</tr>
</thead>
<tbody>
<tr>
<td>(withdraw, deposit, <strong>init</strong>)</td>
<td></td>
</tr>
</tbody>
</table>

Instance attributes of jim_account

| balance | 0 |
| holder   | 'Jim' |
| interest | 0.08 |

Instance attributes of tom_account

| balance | 0 |
| holder   | 'Tom' |

```python
>>> 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

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest = 0.02
>>> jim_account.interest = 0.02
>>> 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
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
Attribute Assignment Statements

Account class attributes

interest: 0.02  0.04  
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

>>> tom_account.interest
0.02
>>> jim_account.interest
0.02

>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> jim_account.interest
0.04

>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
Attribute Assignment Statements

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

`Account` class attributes
- interest: 0.02, 0.04, 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'

```python
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> Account.interest
0.05
```
Attribute Assignment Statements

```python
>>> 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
>>> jim_account.interest = 0.08
>>> Account.interest = 0.05
>>> tom_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

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

```python
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.

```python
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 base class.
Inheritance Example

A CheckingAccount is a specialized type of Account
Inheritance Example

A `CheckingAccount` is a specialized type of `Account`

```python
>>> ch = CheckingAccount('Tom')
```
Inheritance Example

A `CheckingAccount` is a specialized type of `Account`:

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

A `CheckingAccount` is a specialized type of `Account`

```python
>>> 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`

```python
>>> ch = CheckingAccount('Tom')
```

```python
>>> ch.interest  # Lower interest rate for checking accounts
0.01
```

```python
>>> ch.deposit(20)  # Deposits are the same
20
```

```python
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```
Inheritance Example

A `CheckingAccount` is a specialized type of `Account`

```python
>>> ch = CheckingAccount('Tom')
```

```python
>>> ch.interest  # Lower interest rate for checking accounts
0.01
```

```python
>>> ch.deposit(20)  # Deposits are the same
20
```

```python
>>> 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`

```python
>>> ch = CheckingAccount('Tom')
```

```python
>>> ch.interest  # Lower interest rate for checking accounts
0.01
```

```python
>>> ch.deposit(20)  # Deposits are the same
20
```

```python
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```

Most behavior is shared with the base class `Account`

```python
class CheckingAccount(Account):
```
Inheritance Example

A `CheckingAccount` is a specialized type of `Account`

```python
>>> 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`

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

A `CheckingAccount` is a specialized type of `Account`

```python
>>> 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`

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

A CheckingAccount is a specialized type of Account

```python
>>> 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

```python
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`

```python
>>> 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`

```python
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

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

Most behavior is shared with the base class Account

```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)
```
Inheritance Example

A CheckingAccount is a specialized type of Account

```python
>>> 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

```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)
```
Inheritance Example

A `CheckingAccount` is a specialized type of `Account`

```python
>>> 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`

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

A `CheckingAccount` is a specialized type of `Account`

```python
>>> 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`

```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)
```

```python
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 \textit{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.

```python
>>> ch = CheckingAccount('Tom')  # Calls Account.__init__
```
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.

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

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:

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.

```python
>>> 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.

```python
>>> 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.

```python
>>> 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
Don't repeat yourself; use existing implementations

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

Don't repeat yourself; use existing implementations

Attributes that have been overridden are still accessible via class objects

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

Don't repeat yourself; use existing implementations

Attributes that have been overridden are still accessible via class objects

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

Attribute look-up on base class
Designing for Inheritance

Don't repeat yourself; use existing implementations

Attributes that have been overridden are still accessible via class objects

Look up attributes on instances whenever possible

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

Don't repeat yourself; use existing implementations

Attributes that have been overridden are still accessible via class objects

Look up attributes on instances whenever possible

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

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

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

A class may inherit from multiple base classes in Python

CleverBank marketing executive has an idea:
- Low interest rate of 1%
Multiple Inheritance

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

A class may inherit from multiple base classes in Python

CleverBank marketing executive has an idea:
- Low interest rate of 1%
- A $1 fee for withdrawals
Multiple Inheritance

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

A class may inherit from multiple base classes in Python

CleverBank marketing executive has an idea:
• Low interest rate of 1%
• A $1 fee for withdrawals
• A $2 fee for deposits
Multiple Inheritance

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

A class may inherit from multiple base classes in Python

CleverBank marketing executive has an idea:
  • Low interest rate of 1%
  • A $1 fee for withdrawals
  • A $2 fee for deposits
  • A free dollar when you open your account
Multiple Inheritance

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

A class may inherit from multiple base classes in Python

CleverBank marketing executive has an idea:
- Low interest rate of 1%
- A $1 fee for withdrawals
- A $2 fee for deposits
- A free dollar when you open your account

class 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.

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.

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

>>> such_a_deal = AsSeenOnTVAccount('John')
Multiple Inheritance

A class may inherit from multiple base classes in Python.

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

>>> such_a_deal = AsSeenOnTVAccount('John')

>>> such_a_deal.balance
1
Multiple Inheritance

A class may inherit from multiple base classes in Python.

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

>>> 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!
```

```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.

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

>>> 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.

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

>>> such_a_deal = AsSeenOnTVAccount('John')

>>> such_a_deal.balance
1

>>> such_a_deal.deposit(20)
19

>>> such_a_deal.withdraw(5)
13
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
>>> such_a_deal.deposit(20)
19
>>> such_a_deal.withdraw(5)
13
```
Resolving Ambiguous Class Attribute Names

```python
>>> 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

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

Grandma    Grandpa    Grandaddy    Gramammy
Biological Inheritance

Grandma → Grandpa

↓

Mom

↓

Dad

↓

Grandaddy → Gramammy
Biological Inheritance

Grandma  Grandpa  Grandaddy  Gramammy

Mom       Dad

You
Biological Inheritance

Grandma  Grandpa  Grandaddy  Gramammy

Aunt  Mom  Dad  You
Biological Inheritance

- some_guy
  - Grandma
    - Grandpa
      - Grandaddy
      - Gramammy
  - Half
    - Aunt
  - Mom
    - Dad
      - You
Biological Inheritance

some_guy  Grandma  Grandpa  Grandaddy  Gramammy

            Half Aunt   Mom     Dad

some_other_guy  Half Cousin  You  You
Biological Inheritance

- Grandma
- Grandpa
- Gramammy
- Grandaddy
- Aunt
- Mom
- Dad
- You
- Half Aunt
- Mom
- Dad
- Half Cousin
- some_other_guy
- Half Cousin
Biological Inheritance
Biological Inheritance

Grandma  Grandpa  Grandaddy  Gramammy

Double  Half  Aunt  Mom  Dad

some_other_guy

Double  Half Cousin

You

You
Biological Inheritance

Grandma  Grandpa  Grandaddy  Gramammy

Double Half  Aunt  Mom  Dad  Double Half Uncle
	some_other_guy

Double Half Cousin  You
Biological Inheritance

Grandma  Grandpa  Grandaddy  Gramammy

Double Half Aunt  Mom  Dad  Double Half Uncle

Double Half Cousin  You
Biological Inheritance

Grandma  Grandpa  Grandaddy  Gramammy

Double Half Aunt  Mom  Dad  Double Half Uncle

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