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 attributes: attributes of instance objects.
Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs
Classes are objects too, so they have attributes
Instance attributes: attributes of instance objects
Class attributes: attributes of class objects
Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name–value pairs.
Classes are objects too, so they have attributes.
Instance attributes: attributes of instance objects.
Class attributes: attributes of class objects.

Terminology:
Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs.
Classes are objects too, so they have attributes.
Instance attributes: attributes of instance objects.
Class attributes: attributes of class objects.

Terminology:
Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name–value pairs
Classes are objects too, so they have attributes
Instance attributes: attributes of instance objects
Class attributes: attributes of class objects
Terminology: Attributes, Functions, and Methods

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

Classes are objects too, so they have attributes.

Instance attributes: attributes of instance objects.

Class attributes: attributes of class objects.

Terminology:
**Terminology: Attributes, Functions, and Methods**

All objects have attributes, which are name–value pairs.

Classes are objects too, so they have attributes.

**Instance attributes**: attributes of instance objects.

**Class attributes**: attributes of class objects.

**Terminology:**

**Python object system:**

![Diagram showing the relationship between class attributes, methods, and functions.]
Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs.
Classes are objects too, so they have attributes.
Instance attributes: attributes of instance objects.
Class attributes: attributes of class objects.

Terminology:

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 attributes: attributes of instance objects.
Class attributes: attributes of class objects.

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.
Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

Classes are objects too, so they have attributes

Instance attributes: attributes of instance objects

Class attributes: attributes of class objects

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.
Looking Up Attributes by Name (Abbreviated)

<expression> . <name>
Looking Up Attributes by Name (Abbreviated)

\(<expression> . <name>\)

To evaluate a dot expression:
Looking Up Attributes by Name (Abbreviated)

\(<\text{expression}>\ . \ <\text{name}>\)

To evaluate a dot expression:

1. Evaluate the \(<\text{expression}>\).
Looking Up Attributes by Name (Abbreviated)

<expression>. <name>

To evaluate a dot expression:

1. Evaluate the <expression>.

2. <name> is matched against the instance attributes.
Looking Up Attributes by Name (Abbreviated)

To evaluate a dot expression:

1. Evaluate the \(<expression>\).
2. \(<name>\) is matched against the instance attributes.
3. If not found, \(<name>\) is looked up in the class.
Looking Up Attributes by Name (Abbreviated)

To evaluate a dot expression:

1. Evaluate the `<expression>`.

2. `<name>` is matched against the instance attributes.

3. If not found, `<name>` is looked up in the class.

4. That class attribute value is returned unless it is a function, in which case a bound method is returned.
Looking Up Attributes by Name (Abbreviated)

To evaluate a dot expression:

1. Evaluate the `<expression>`.

2. `<name>` is matched against the instance attributes.

3. If not found, `<name>` is looked up in the class.

4. That class attribute value is returned unless it is a function, in which case a bound method is returned.
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(object):
    interest = 0.02  # Class attribute

    def __init__(self, account_holder):
        self.balance = 0  # Instance attribute
        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(object):
    interest = 0.02          # Class attribute

    def __init__(self, account_holder):
        self.balance = 0       # Instance attribute
        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.

class Account(object):
    interest = 0.02  # Class attribute

    def __init__(self, account_holder):
        self.balance = 0  # Instance attribute
        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.

class Account(object):
    interest = 0.02  # Class attribute
    def __init__(self, account_holder):
        self.balance = 0  # Instance attribute
        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(object):
    interest = 0.02  # Class attribute

    def __init__(self, account_holder):
        self.balance = 0  # Instance attribute
        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
```
Class Attributes

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

class Account(object):
    interest = 0.02  # Class attribute

    def __init__(self, account_holder):
        self.balance = 0  # Instance attribute
        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

*interest* is not part of the instance that was somehow copied from the class!
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 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
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

```
 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

```
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

```
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 "interest" attribute of tom_account
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

```
tom_account.interest = 0.08
```

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

Attribute assignment statement adds or modifies the "interest" attribute 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

Instance Attribute Assignment

```
tom_account.interest = 0.08
```

This expression evaluates to an object

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

Class Attribute Assignment

```
Account.interest = 0.04
```

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

interest: 0.02
Account class attributes

interest: 0.02
Account class attributes

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__)

balance: 0
holder: 'Jim'

>>> jim_account = Account('Jim')
Attribute Assignment Statements

Account class attributes

interest: 0.02
(withdraw, deposit, __init__)

balance: 0
holder: 'Jim'

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

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

Account class attributes

balance: 0
holder: 'Jim'

balance: 0
holder: 'Tom'

interest: 0.02
(withdraw, deposit, __init__)

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

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

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

Account class attributes

- **balance**: 0
- **holder**: 'Jim'

balance: 0
holder: 'Jim'

Interest: 0.02
(withdraw, deposit, __init__)

balance: 0
holder: 'Tom'

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

Account class attributes

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

interest: 0.02 0.04 (withdraw, deposit, __init__)
Attribute Assignment Statements

Account class attributes

- **interest**: 0.02 0.04 (withdraw, deposit, __init__)
- **balance**: 0
- **holder**: 'Jim'
- **balance**: 0
- **holder**: 'Tom'

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

Account class attributes

balance: 0
holder: 'Jim'

balance: 0
holder: 'Tom'

interest: 0.02 0.04 (withdraw, deposit, __init__)

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

Account class attributes

| balance: 0 | balance: 0 |
| holder: 'Jim' | holder: 'Tom' |
| interest: 0.08 | interest: 0.02 |

interest: 0.02 0.04 (withdraw, deposit, __init__)

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

Account class attributes

balance: 0
holder: 'Jim'
interest: 0.08

balance: 0
holder: 'Tom'

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

interest: 0.02 0.04
(withdraw, deposit, __init__)
Attribute Assignment Statements

Account class attributes

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

Account

interest: 0.02 0.04
(withdraw, deposit, __init__)

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

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

Account class attributes

interest: 0.02 0.04
(withdraw, deposit, __init__)

balance: 0
holder: 'Jim'
interest: 0.08

balance: 0
holder: 'Tom'

>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
>>> tom_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
Attribute Assignment Statements

Account class attributes

```python
>>> jim_account = Account('Jim')
```
```
balance: 0
holder: 'Jim'
interest: 0.08
```
```
>>> tom_account = Account('Tom')
```
```
balance: 0
holder: 'Tom'
interest: 0.02
```

```python
>>> jim_account = Account('Jim')
```
```
balance: 0
holder: 'Jim'
interest: 0.02
```
```
>>> tom_account = Account('Tom')
```
```
balance: 0
holder: 'Tom'
interest: 0.04
```

```python
>>> jim_account = Account('Jim')
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account = Account('Tom')
>>> Account.interest = 0.04
>>> tom_account = Account('Tom')
```
```
>>> jim_account = Account('Jim')
>>> jim_account = Account('Jim')
>>> Account.interest = 0.05
```
```
>>> Account.interest = 0.04
>>> Account.interest = 0.05
```
```
>>> Account.interest = 0.05
```
```
>>> Account.interest = 0.05
```
```
>>> Account.interest = 0.05
```
```
>>> Account.interest = 0.05
```
Attribute Assignment Statements

Account class attributes

balance: 0
holder: 'Jim'
interest: 0.08

balance: 0
holder: 'Tom'

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

interest: 0.02 0.04 0.05
(withdraw, deposit, __init__)
Attribute Assignment Statements

```python
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest = 0.02
>>> jim_account.interest = 0.04
>>> Account.interest = 0.05
>>> jim_account.interest = 0.08
>>> tom_account.interest = 0.04
>>> Account.interest = 0.04
>>> tom_account.interest = 0.04
```

Account class attributes

<table>
<thead>
<tr>
<th></th>
<th>balance: 0</th>
<th>holder: 'Jim'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>balance: 0</td>
<td>holder: 'Tom'</td>
</tr>
<tr>
<td></td>
<td>interest: 0.02</td>
<td></td>
</tr>
<tr>
<td></td>
<td>interest: 0.04</td>
<td></td>
</tr>
<tr>
<td></td>
<td>interest: 0.08</td>
<td></td>
</tr>
</tbody>
</table>
Inheritance
Inheritance

A technique for relating classes together
Inheritance

A technique for relating classes together

Common use: Similar classes differ in amount of specialization
Inheritance

A technique for relating classes together

Common use: Similar classes differ in amount of specialization

Two classes have overlapping attribute sets, but one represents a special case of the other.
Inheritance

A technique for relating classes together

Common use: Similar classes differ in amount of specialization

Two classes have overlapping attribute sets, but one represents a special case of the other.

    class <name>(<base class>):
        <suite>
Inheritance

A technique for relating classes together

Common use: Similar classes differ in amount of specialization

Two classes have overlapping attribute sets, but one represents a special case of the other.

```python
class <name>(<base class>):
  <suite>
```

Conceptually, the new subclass "shares" attributes with its base class.
Inheritance

A technique for relating classes together

Common use: Similar classes differ in amount of specialization

Two classes have overlapping attribute sets, but one represents a special case of the other.

```python
class <name>(<base class>):
    <suite>
```

Conceptually, the new subclass "shares" attributes with its base class.

The subclass may *override* certain inherited attributes.
Inheritance

A technique for relating classes together

Common use: Similar classes differ in amount of specialization

Two classes have overlapping attribute sets, but one represents a special case of the other.

```python
class <name>(<base class>):
    <suite>
```

Conceptually, the new subclass "shares" attributes with its base class.

The subclass may override certain inherited attributes.

Using inheritance, we implement a subclass by specifying its difference 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.

```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')
>>> 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.

```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
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 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')
```

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

```python
>>> 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.

```python
>>> 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.

```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.

>>> 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
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.

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)
Designing for Inheritance: General Base Classes
Base classes may contain logic that is meant for subclasses.
Designing for Inheritance: General Base Classes

Base classes may contain logic that is meant for subclasses.

Example: Same CheckingAccount behavior; different approach

Demo
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 Account instances 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 Account instances as an attribute.

No local state at all? Just write a pure function!
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 wants:
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 wants:
  - 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 wants:

- 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 wants:
   • 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 wants:
  • 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 wants:
  • 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!

Instance attribute

>>> 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
>> 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
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
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

Account

CheckingAccount  SavingsAccount

AsSeenOnTVAccount

>>> such_a_deal = AsSeenOnTVAccount("John")
>>> such_a_deal.balance
1
>>> such_a_deal.deposit(20)
19
>>> such_a_deal.withdraw(5)
13
Human Relationships
Human Relationships

Grandma  Grandpa  Grandaddy  Gramammy
Human Relationships

Grandma → Grandpa

Mom → Grandaddy

Dad → Gramammy
Human Relationships

- Grandma
- Grandpa
- Grandaddy
- Gramammy

- Mom
- Dad

- You
Human Relationships

- Grandma
- Grandpa
- Gramammy
- Grandaddy
- Aunt
- Mom
- Dad
- You
Human Relationships

Some_Guy  Grandma  Grandpa  Grandaddy  Gramammy

Aunt  Mom  Dad  You
Human Relationships

Some_Guy  Grandma  Grandpa  Grandaddy  Gramammy

Half  Aunt  Mom  Dad

You
Human Relationships

- Some_Guy
- Grandma
- Grandpa
- Grandaddy
- Gramammy
- Aunt
- Mom
- Dad
- You
- Half
- Half Cousin
- Some_Dude
Human Relationships

Grandma  Grandpa  Grandaddy  Gramammy

Half Aunt  Mom  Dad

Some_Dude  Half Cousin  You
Human Relationships

Grandma  Grandpa  Grandaddy  Gramammy

Double Half Aunt  Mom  Dad

Half Cousin  You

Some_Dude  You
Human Relationships

- Grandma
- Grandpa
- Gramammy
- Grandaddy
- Mom
- Dad
- You
- Aunt
- Some_Dude
- Double Half Cousin
- Double Half Aunt
Human Relationships

- Grandma
- Grandpa
- Gramammy
- Grandaddy
- Aunt
- Mom
- Dad
- Double Half Uncle
- Double Half Cousin
- Some_Dude
- Double Half Aunt
- You
Human Relationships

Grandma
Grandpa
Gramammy
Grandaddy

Double Half Aunt
Mom
Dad
Double Half Uncle

Double Half Cousin
You