Announcements

- HW7 due on Wednesday
- Ants project out
Dot Expressions
Objects receive messages via dot notation
Dot Expressions

Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class
Dot Expressions

Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class

<expression> . <name>
Dot Expressions

Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class

<expression> . <name>

The <expression> can be any valid Python expression
Dot Expressions

Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class

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

The \(<\text{expression}>\) can be any valid Python expression

The \(<\text{name}>\) must be a simple name
Dot Expressions

Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class

\(<expression> . <name>\)

The \(<expression>\) can be any valid Python expression

The \(<name>\) must be a simple name

Evaluates to the value of the attribute \(\text{looked up by} <name>\)
in the object that is the value of the \(<expression>\)
Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class

<expression> . <name>

The <expression> can be any valid Python expression

The <name> must be a simple name

Evaluates to the value of the attribute **looked up** by <name> in the object that is the value of the <expression>

```
tom_account.deposit(10)
```
Dot Expressions

Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class

<expression> . <name>

The <expression> can be any valid Python expression

The <name> must be a simple name

Evaluates to the value of the attribute looked up by <name> in the object that is the value of the <expression>

```
tom_account.deposit(10)
```
Dot Expressions

Objects receive messages via dot notation

Dot notation accesses attributes of the instance or its class

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

The \(<\text{expression}>\) can be any valid Python expression

The \(<\text{name}>\) must be a simple name

Evaluates to the value of the attribute \textit{looked up} by \(<\text{name}>\) in the object that is the value of the \(<\text{expression}>\)

\texttt{tom\_account.deposit(10)}

\textbf{Dot expression}

\textbf{Call expression}
Accessing Attributes
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary.
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary

```python
>>> getattr(tom_account, 'balance')
```
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary

```python
>>> getattr(tom_account, 'balance')
10
```
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary:

```python
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
```
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary

```python
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
True
```
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary

```python
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
True
```

`getattr` and dot expressions look up a name in the same way
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary

```python
>>> getattr(tom_account, 'balance')
10
```

```python
>>> hasattr(tom_account, 'deposit')
True
```

`getattr` and dot expressions look up a name in the same way

Looking up an attribute name in an object may return:
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary

```python
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
True
```

`getattr` and dot expressions look up a name in the same way

Looking up an attribute name in an object may return:

- One of its instance attributes, **or**
Accessing Attributes

Using `getattr`, we can look up an attribute using a string, just as we did with a dispatch function/dictionary:

```python
>>> getattr(tom_account, 'balance')
10

>>> hasattr(tom_account, 'deposit')
True
```

`getattr` and dot expressions look up a name in the same way.

Looking up an attribute name in an object may return:

- One of its instance attributes, **or**
- One of the attributes of its class
Methods and Functions

Python distinguishes between:
Python distinguishes between:

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

\[
\text{Object} + \ \text{Function} = \ \text{Bound Method}
\]
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.

\[
\text{Object} \; + \; \text{Function} \; = \; \text{Bound Method}
\]

\[\text{>>> type(Account.deposit)}\]
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.

\[
\text{Object} + \text{Function} = \text{Bound Method}
\]

```python
>>> 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.

\[
\text{Object} + \text{Function} = \text{Bound Method}
\]

```python
>>> 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.

\[
\text{Object} + \text{Function} = \text{Bound Method}
\]

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

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

\[
\text{Object} + \text{Function} = \text{Bound Method}
\]

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

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

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

\[
\text{Object + Function} = \text{Bound Method}
\]

```python
>>> 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.

\[ \text{Object} + \text{Function} = \text{Bound Method} \]

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

>>> Account.deposit(tom_account, 1001)
1011
>>> tom_account.deposit(1000)
```
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

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

>>> Account.deposit(tom_account, 1001)
1011
>>> tom_account.deposit(1000)
2011
```
Methods and Currying
Methods and Currying

Earlier, we saw *currying*, which converts a function that takes in multiple arguments into multiple chained functions.
Earlier, we saw *currying*, which converts a function that takes in multiple arguments into multiple chained functions.

```python
def curry(f):
    def outer(x):
        def inner(*args):
            return f(x, *args)
        return inner
    return outer
```
Methods and Currying

Earlier, we saw currying, which converts a function that takes in multiple arguments into multiple chained functions.

```python
def curry(f):
    def outer(x):
        def inner(*args):
            return f(x, *args)
        return inner
    return outer

>>> add2 = curry(add)(2)
>>> add2(3)
5
```
Methods and Currying

Earlier, we saw *currying*, which converts a function that takes in multiple arguments into multiple chained functions.

The same procedure can be used to create a bound method from a function

```python
def curry(f):
    def outer(x):
        def inner(*args):
            return f(x, *args)
        return inner
    return outer

>>> add2 = curry(add)(2)
>>> add2(3)
5
```
Methods and Currying

Earlier, we saw *currying*, which converts a function that takes in multiple arguments into multiple chained functions.

The same procedure can be used to create a bound method from a function

```python
def curry(f):
    def outer(x):
        def inner(*args):
            return f(x, *args)
        return inner
    return outer
```

```python
>>> add2 = curry(add)(2)
>>> add2(3)
5
```

```python
>>> tom_deposit = curry(Account.deposit)(tom_account)
>>> tom_deposit(1000)
3011
```
Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs.
Attributes, Functions, and Methods

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

Classes are objects too, so they have attributes.
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
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.
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:
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:
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:

Class Attributes

Methods

Python object system:

Functions
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.
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.
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 on instances evaluate to bound methods for class attributes that are functions.
Looking Up Attributes by Name

\(<\text{expression}>\ . \ <\text{name}>\)
Looking Up Attributes by Name

\(<expression> . <name>\)

To evaluate a dot expression:
Looking Up Attributes by Name

\[ \text{<expression>} . \text{<name>} \]

To evaluate a dot expression:

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

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

To evaluate a dot expression:

1. Evaluate the \(<\text{expression}>\).

2. \(<\text{name}>\) is matched against the instance attributes.
Looking Up Attributes by Name

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

\(<expression> . <name>\)

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

<expression> . <name>

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

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.

```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')
```
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
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
```
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
```

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
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 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
```
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
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 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
Attribute Assignment Statements

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

```python
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

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

balance: 0
holder: 'Jim'

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

Account class attributes
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')

Account class attributes

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

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

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

Account class attributes

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

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
>>> jim_account.interest

Account class attributes

balance: 0
holder: 'Jim'

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

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
>>> jim_account.interest
0.02
>>> tom_account.interest
```
Attribute Assignment Statements

Account class attributes

balance: 0
holder: 'Jim'

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
```
>>> 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

```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

```
>>> 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
0.02
```

**Account class attributes**

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

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

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

```python
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
>>> 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
>>> tom_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
```

Account class attributes

| balance:    | 0     |
| holder:     | 'Jim' |

| 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
>>> tom_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
>>> tom_account.interest
0.02

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

balance: 0
holder: 'Jim'
interest: 0.08
```

```python
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.02

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
>>> tom_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
```
Attribute Assignment Statements

Account class attributes

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

>>> 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__)

>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_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
>>> 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
>>> tom_account.interest
0.02
>>> Account.interest = 0.04
>>> 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
>>> 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

```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
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
```

**Account class attributes**

| balance: 0 | balance: 0 |
| holder: 'Jim' | holder: 'Tom' |
| interest: 0.08 | interest: 0.00 |

(withdraw, deposit, __init__)
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

>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05

Account class attributes

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

balance: 0
holder: 'Jim'
interest: 0.08

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
>>> tom_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_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
>>> tom_account.interest
0.02
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
>>> jim_account.interest
0.08
```
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.

```python
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 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
```
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
```
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
```
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
```
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):
```
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
def __init__(self, name):
    self.name = name
    self.balance = 0

    # A bank account that charges for withdrawals.
    self.withdraw_fee = 1
    self.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**

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,
```
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)
```
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)
```
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.

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

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.
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)
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)
General Base Classes
General Base Classes

Base classes may contain logic that is meant for subclasses.
Base classes may contain logic that is meant for subclasses.

Example: Same CheckingAccount behavior; different approach
Base classes may contain logic that is meant for subclasses.

Example: Same `CheckingAccount` behavior; different approach

```python
class Account(object):
```
Base classes may contain logic that is meant for subclasses.

Example: Same `CheckingAccount` behavior; different approach

```python
class Account(object):
    interest = 0.02
```
Base classes may contain logic that is meant for subclasses.

Example: Same `CheckingAccount` behavior; different approach

```python
class Account(object):
    interest = 0.02

    def withdraw(self, amount):
```
Base classes may contain logic that is meant for subclasses.

Example: Same **CheckingAccount** behavior; different approach

```python
class Account(object):
    interest = 0.02

    def withdraw(self, amount):
        if amount > self.balance:
            # Code here
```
Base classes may contain logic that is meant for subclasses.

Example: Same **CheckingAccount** behavior; different approach

```python
class Account(object):
    interest = 0.02

    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
```
General Base Classes

Base classes may contain logic that is meant for subclasses.

Example: Same `CheckingAccount` behavior; different approach

```python
class Account(object):
    interest = 0.02

    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
```
General Base Classes

Base classes may contain logic that is meant for subclasses.

Example: Same `CheckingAccount` behavior; different approach

```python
class Account(object):
    interest = 0.02

    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```
General Base Classes

Base classes may contain logic that is meant for subclasses.

Example: Same `CheckingAccount` behavior; different approach

```python
class Account(object):
    interest = 0.02
    withdraw_fee = 0
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```
General Base Classes

Base classes may contain logic that is meant for subclasses.

Example: Same `CheckingAccount` behavior; different approach

```python
class Account(object):
    interest = 0.02
    withdraw_fee = 0

def withdraw(self, amount):
    amount += self.withdraw_fee
    if amount > self.balance:
        return 'Insufficient funds'
    self.balance = self.balance - amount
    return self.balance
```
Base classes may contain logic that is meant for subclasses.

Example: Same `CheckingAccount` behavior; different approach

```python
class Account(object):
    interest = 0.02
    withdraw_fee = 0
    def withdraw(self, amount):
        amount += self.withdraw_fee
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance

class CheckingAccount(Account):
```

```python
```
General Base Classes

Base classes may contain logic that is meant for subclasses.

Example: Same `CheckingAccount` behavior; different approach

```python
class Account(object):
    interest = 0.02
    withdraw_fee = 0
    def withdraw(self, amount):
        amount += self.withdraw_fee
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance

class CheckingAccount(Account):
    interest = 0.01
```
General Base Classes

Base classes may contain logic that is meant for subclasses.

Example: Same `CheckingAccount` behavior; different approach

```python
class Account(object):
    interest = 0.02
    withdraw_fee = 0

def withdraw(self, amount):
    amount += self.withdraw_fee
    if amount > self.balance:
        return 'Insufficient funds'
    self.balance = self.balance - amount
    return self.balance

class CheckingAccount(Account):
    interest = 0.01
    withdraw_fee = 1
```
Base classes may contain logic that is meant for subclasses.

Example: Same `CheckingAccount` behavior; different approach

```python
class Account(object):
    interest = 0.02
    withdraw_fee = 0

def withdraw(self, amount):
    amount += self.withdraw_fee
    if amount > self.balance:
        return 'Insufficient funds'
    self.balance = self.balance - amount
    return self.balance

class CheckingAccount(Account):
    interest = 0.01
    withdraw_fee = 1
```

May be overridden by subclasses.
Base classes may contain logic that is meant for subclasses.

Example: Same **CheckingAccount** behavior; different approach

```python
class Account(object):
    interest = 0.02
    withdraw_fee = 0

def withdraw(self, amount):
    amount += self.withdraw_fee
    if amount > self.balance:
        return 'Insufficient funds'
    self.balance = self.balance - amount
    return self.balance
```

```python
class CheckingAccount(Account):
    interest = 0.01
    withdraw_fee = 1
```

May be overridden by subclasses

Nothing else needed in this class
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
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

```
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:
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

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
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

```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
• 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.

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")
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!

>>> such_a_deal = AsSeenOnTVAccount("John")
>>> such_a_deal.balance
```

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!

>>> such_a_deal = AsSeenOnTVAccount("John")
>>> such_a_deal.balance
```
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
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)
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)
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

Instance attribute
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

Instance attribute
SavingsAccount method
CheckingAccount method
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

```python
>>> 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

<table>
<thead>
<tr>
<th>Grandma</th>
<th>Grandpa</th>
<th>Grandaddy</th>
<th>Gramammy</th>
</tr>
</thead>
</table>

Human Relationships

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

Grandma  Grandpa  Grandaddy  Gramammy

Mom  Dad

You
Human Relationships

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

Grandma  Grandpa  Granddaddy  Gramammy

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Aunt</td>
<td>Mom</td>
<td>Dad</td>
<td>You</td>
</tr>
</tbody>
</table>

You
Human Relationships

Grandma  Grandpa  Granddaddy  Gramammy
Aunt      Mom       Dad           You
Human Relationships

Some_Guy → Grandma → Aunt
Some_Guy → Grandpa → Mom
Grandma → Grandaddy → Dad
Grandpa → Gramammy

You
Human Relationships

Some_Guy → Grandma → Grandpa → Gramammy
Half → Aunt → Mom → Dad → You
Grandaddy → You
Human Relationships

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

Some_Guy

Grandma

Grandpa

Grandaddy

Gramammy

Aunt

Half

Mom

Dad

Some_Dude

Half Cousin

You
Human Relationships

Grandma  Grandpa  Grandaddy  Gramammy

Half  Aunt  Mom  Dad  You

Some_Dude  Half Cousin
Human Relationships

Grandma  Grandpa  Grandaddy  Gramammy

Double  Half  Aunt  Mom  Dad

Some_Dude  Half Cousin  You
Human Relationships

Grandma  Grandpa  Grandaddy  Gramammy

Double  Half  Aunt  Mom  Dad

Some_Dude

Double  Half Cousin  You  You
Human Relationships

Grandma  Grandpa  Grandaddy  Gramammy

Double  Half  Aunt  Mom  Dad  Double Half Uncle

Some_Dude

Double  Half Cousin  You
Human Relationships

Grandma → Grandpa → Grandaddy → Gramammy → Dad → Double Half Uncle

Mom → You

Aunt → Double Half Uncle

Half Cousin → Double Half Uncle

Double → Half → Aunt

Some_Dude → Double Half Uncle
Human Relationships

Grandma → Grandpa → Gramammy

Grandaddy

Double Mom Dad

Half Uncle

Double Half Cousin

You
Human Relationships

Grandma → Grandpa → Grandaddy → Gramammy

Double → Half ← Aunt ← Mom ← Dad ← Double Half Uncle

Quad Double → Half Cousin ← You
Human Relationships

Grandma → Grandpa
Grandaddy → Gramammy

Double Half Aunt
Half Cousin

Quadruple

Mom → You
Dad → Double Half Uncle