61A Lecture 14

Wednesday, February 25
Announcements

• Project 2 due Thursday 2/26 @ 11:59pm
Announcements

• Project 2 due Thursday 2/26 @ 11:59pm

  ▪ Extra office hours on Wednesday 2/25 4pm–6pm in Bechtel (Garbarini Lounge)
Announcements

• Project 2 due Thursday 2/26 @ 11:59pm
  • Extra office hours on Wednesday 2/25 4pm–6pm in Bechtel (Garbarini Lounge)
  • Bonus point for early submission by Wednesday 2/25 @ 11:59pm!
Announcements

• Project 2 due Thursday 2/26 @ 11:59pm
  ▪ Extra office hours on Wednesday 2/25 4pm–6pm in Bechtel (Garbarini Lounge)
  ▪ Bonus point for early submission by Wednesday 2/25 @ 11:59pm!
• Relocated office hours on Thursday 2/26: 380 Soda (11am–3pm) & 606 Soda (3pm–6pm)
Object-Oriented Programming
Object-Oriented Programming
Object-Oriented Programming

A method for organizing programs
Object-Oriented Programming

A method for organizing programs

• Data abstraction
Object-Oriented Programming

A method for organizing programs

- Data abstraction
- Bundling together information and related behavior
Object-Oriented Programming

A method for organizing programs

• Data abstraction
• Bundling together information and related behavior

A metaphor for computation using distributed state
Object-Oriented Programming

A method for organizing programs

• Data abstraction
• Bundling together information and related behavior

A metaphor for computation using distributed state

• Each object has its own local state
Object-Oriented Programming

A method for organizing programs

• Data abstraction
• Bundling together information and related behavior

A metaphor for computation using distributed state

• Each object has its own local state
• Each object also knows how to manage its own local state, based on method calls
Object-Oriented Programming

A method for organizing programs

- Data abstraction
- Bundling together information and related behavior

A metaphor for computation using distributed state

- Each object has its own local state
- Each object also knows how to manage its own local state, based on method calls
- Method calls are messages passed between objects
Object-Oriented Programming

A method for organizing programs

• Data abstraction
• Bundling together information and related behavior

A metaphor for computation using distributed state

• Each object has its own local state
• Each object also knows how to manage its own local state, based on method calls
• Method calls are messages passed between objects
• Several objects may all be instances of a common type
Object-Oriented Programming

A method for organizing programs

- Data abstraction
- Bundling together information and related behavior

A metaphor for computation using distributed state

- Each object has its own local state
- Each object also knows how to manage its own local state, based on method calls
- Method calls are messages passed between objects
- Several objects may all be instances of a common type
- Different types may relate to each other
Object-Oriented Programming

A method for organizing programs

• Data abstraction
• Bundling together information and related behavior

A metaphor for computation using distributed state

• Each object has its own local state
• Each object also knows how to manage its own local state, based on method calls
• Method calls are messages passed between objects
• Several objects may all be instances of a common type
• Different types may relate to each other

Specialized syntax & vocabulary to support this metaphor
Object-Oriented Programming

A method for organizing programs
- Data abstraction
- Bundling together information and related behavior

A metaphor for computation using distributed state
- Each object has its own local state
- Each object also knows how to manage its own local state, based on method calls
- Method calls are messages passed between objects
- Several objects may all be instances of a common type
- Different types may relate to each other

Specialized syntax & vocabulary to support this metaphor
Object-Oriented Programming

A method for organizing programs
• Data abstraction
• Bundling together information and related behavior

A metaphor for computation using distributed state
• Each object has its own local state
• Each object also knows how to manage its own local state, based on method calls
• Method calls are messages passed between objects
• Several objects may all be instances of a common type
• Different types may relate to each other

Specialized syntax & vocabulary to support this metaphor
Object-Oriented Programming

A method for organizing programs
• Data abstraction
• Bundling together information and related behavior

A metaphor for computation using distributed state
• Each object has its own local state
• Each object also knows how to manage its own local state, based on method calls
• Method calls are messages passed between objects
• Several objects may all be instances of a common type
• Different types may relate to each other

Specialized syntax & vocabulary to support this metaphor

John's Account

John

Steven's Account
Object-Oriented Programming

A method for organizing programs

- Data abstraction
- Bundling together information and related behavior

A metaphor for computation using distributed state

- Each object has its own local state
- Each object also knows how to manage its own local state, based on method calls
- Method calls are messages passed between objects
- Several objects may all be instances of a common type
- Different types may relate to each other

Specialized syntax & vocabulary to support this metaphor
Object-Oriented Programming

A method for organizing programs
• Data abstraction
• Bundling together information and related behavior

A metaphor for computation using distributed state
• Each object has its own local state
• Each object also knows how to manage its own local state, based on method calls
• Method calls are messages passed between objects
• Several objects may all be instances of a common type
• Different types may relate to each other

Specialized syntax & vocabulary to support this metaphor
Object-Oriented Programming

A method for organizing programs
- Data abstraction
- Bundling together information and related behavior

A metaphor for computation using distributed state
- Each object has its own local state
- Each object also knows how to manage its own local state, based on method calls
- Method calls are messages passed between objects
- Several objects may all be instances of a common type
- Different types may relate to each other

Specialized syntax & vocabulary to support this metaphor
Object-Oriented Programming

A method for organizing programs
• Data abstraction
• Bundling together information and related behavior

A metaphor for computation using distributed state
• Each object has its own local state
• Each object also knows how to manage its own local state, based on method calls
• Method calls are messages passed between objects
• Several objects may all be instances of a common type
• Different types may relate to each other

Specialized syntax & vocabulary to support this metaphor
Classes
A class serves as a template for its instances.
Classes

A class serves as a template for its instances.

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.
A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
```
A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.  

```python
g>>> a = Account('Jim')```
Classes

A class serves as a template for its instances.

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
```
A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
```
A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```
A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

**Idea:** All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.
A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
>>> a.deposit(15)
15
```

**Idea:** All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.
**Classes**

A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the `Account` class should add those attributes to each newly created instance.

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
```

**Idea:** All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.
Classes

A class serves as a template for its instances.

**Idea:** All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

**Idea:** All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.

```
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.balance
5
```
Classes

A class serves as a template for its instances.

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0

>>> a.deposit(15)
15
>>> a.withdraw(10)
5
>>> a.balance
5
>>> a.withdraw(10)
'Insufficient funds'

Idea: All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.
Classes

A class serves as a template for its instances.

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each newly created instance.

Idea: All bank accounts should have "withdraw" and "deposit" behaviors that all work in the same way.

Better idea: All bank accounts share a "withdraw" method and a "deposit" method.
Class Statements
The Class Statement
class <name>:
    <suite>
The Class Statement

class <name>:
    <suite>

A class statement creates a new class and binds that class to <name> in the first frame of the current environment.
The Class Statement

class <name>:
    <suite>

A class statement creates a new class and binds that class to <name> in the first frame of the current environment.

Assignment & def statements in <suite> create attributes of the class (not names in frames)
The Class Statement

```python
class <name>:
    <suite>
```

A class statement creates a new class and binds that class to `<name>` in the first frame of the current environment.

Assignment & def statements in `<suite>` create attributes of the class (not names in frames)
The Class Statement

A class statement creates a new class and binds that class to `<name>` in the first frame of the current environment.

Assignment & def statements in `<suite>` create attributes of the class (not names in frames)

```python
>>> class Clown:
...    nose = 'big and red'
...    def dance():
...        return 'No thanks'
...```
The Class Statement

class <name>:
    <suite>

The suite is executed when the class statement is executed.

A class statement creates a new class and binds that class to <name> in the first frame of the current environment.

Assignment & def statements in <suite> create attributes of the class (not names in frames)

```python
>>> class Clown:
...    nose = 'big and red'
...    def dance():
...        return 'No thanks'
...    
>>> Clown.nose
'big and red'
```
The Class Statement

class <name>:
    <suite>

A class statement creates a new class and binds that class to <name> in the first frame of the current environment.

Assignment & def statements in <suite> create attributes of the class (not names in frames)

```python
>>> class Clown:
...    nose = 'big and red'
...    def dance():
...        return 'No thanks'
... >>> Clown.nose
'big and red'
>>> Clown.dance()
'No thanks'
```
The Class Statement

A class statement creates a new class and binds that class to `<name>` in the first frame of the current environment.

Assignment & def statements in `<suite>` create attributes of the class (not names in frames)

```python
>>> class Clown:
...     nose = 'big and red'
...     def dance():
...         return 'No thanks'
... >>> Clown.nose
'big and red'
>>> Clown.dance()
'No thanks'
>>> Clown
<class '__main__.Clown'>
```
Object Construction
Object Construction

**Idea:** All bank accounts have a *balance* and an account *holder*; the `Account` class should add those attributes to each of its instances

```python
>>> a = Account('Jim')
```
Object Construction

**Idea:** All bank accounts have a `balance` and an account `holder`; the `Account` class should add those attributes to each of its instances

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

When a class is called:
Object Construction

**Idea:** All bank accounts have a *balance* and an account *holder*; the `Account` class should add those attributes to each of its instances

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

When a class is called:

1. A new instance of that class is created:
Object Construction

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each of its instances

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

When a class is called:

1. A new instance of that class is created:
Object Construction

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each of its instances.

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

When a class is called:

1. A new instance of that class is created:

2. The __init__ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression.
When a class is called:

1. A new instance of that class is created:

2. The \texttt{__init__} method of the class is called with the new object as its first argument (named \texttt{self}), along with any additional arguments provided in the call expression.

```python
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```
Object Construction

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each of its instances

>>> a = Account('Jim')

class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
Object Construction

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each of its instances

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

When a class is called:

1. A new instance of that class is created:

2. The `__init__` method of the class is called with the new object as its first argument (named `self`), along with any additional arguments provided in the call expression.

```python
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```
Object Construction

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each of its instances.

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

When a class is called:

1. A new instance of that class is created:
2. The __init__ method of the class is called with the new object as its first argument (named `self`), along with any additional arguments provided in the call expression.

```python
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```
Object Construction

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each of its instances

>>> a = Account('Jim')

When a class is called:

1. A new instance of that class is created:

2. The __init__ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression.

```python
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```
Object Construction

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each of its instances

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

When a class is called:

1. A new instance of that class is created:

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

2. The __init__ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression.
Object Construction

Idea: All bank accounts have a **balance** and an account **holder**; the **Account** class should add those attributes to each of its instances

```python
>>> a = Account('Jim')
>>> a.holder
'Jim'
```

When a class is called:

1. A new instance of that class is created:

2. The **__init__** method of the class is called with the new object as its first argument (named `self`), along with any additional arguments provided in the call expression.

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

__init__ is called a constructor
Object Construction

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each of its instances.

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

>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

When a class is called:

1. A new instance of that class is created:

2. The __init__ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression.
Object Identity
Object Identity

Every object that is an instance of a user-defined class has a unique identity:
Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```python
>>> a = Account('Jim')
>>> b = Account('Jack')
```
Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```python
>>> a = Account('Jim')
>>> b = Account('Jack')
```

Every call to `Account` creates a new `Account` instance. There is only one `Account` class.
Every object that is an instance of a user-defined class has a unique identity:

```python
>>> a = Account('Jim')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Every call to Account creates a new Account instance. There is only one Account class.
Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```python
>>> a = Account('Jim')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Identity operators "is" and "is not" test if two expressions evaluate to the same object:

Every call to Account creates a new Account instance. There is only one Account class.
Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```python
>>> a = Account('Jim')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Identity operators "is" and "is not" test if two expressions evaluate to the same object:

```python
>>> a is a
True
>>> a is not b
True
```
Every object that is an instance of a user-defined class has a unique identity:

```python
>>> a = Account('Jim')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Identity operators "is" and "is not" test if two expressions evaluate to the same object:

```python
>>> a is a
True
>>> a is not b
True
```

Binding an object to a new name using assignment does not create a new object:
Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```python
>>> a = Account('Jim')
>>> b = Account('Jack')
>>> a.balance
0
>>> b.holder
'Jack'
```

Identity operators "is" and "is not" test if two expressions evaluate to the same object:

```python
>>> a is a
True
>>> a is not b
True
```

Binding an object to a new name using assignment does not create a new object:

```python
>>> c = a
>>> c is a
True
```
Methods
Methods
Methods

Methods are functions defined in the suite of a class statement
Methods

Methods are functions defined in the suite of a class statement

class Account:
Methods

Methods are functions defined in the suite of a class statement

```python
def __init__(self, account_holder):
```
Methods

Methods are functions defined in the suite of a class statement

```python
self.balance = 0
```
Methods

Methods are functions defined in the suite of a class statement

```python
self.holder = account_holder
```
Methods

Methods are functions defined in the suite of a class statement

```python
def deposit(self, amount):
```
Methods

Methods are functions defined in the suite of a class statement

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

def deposit(self, amount):

self should always be bound to an instance of the Account class
Methods

Methods are functions defined in the suite of a class statement

```python
self should always be bound to an instance of the Account class

self.balance = self.balance + amount
```
Methods

Methods are functions defined in the suite of a class statement

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

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

*self should always be bound to an instance of the Account class*

```
return self.balance
```
Methods

Methods are functions defined in the suite of a class statement

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

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        self.balance = self.balance - amount
        return self.balance
```

*`self` should always be bound to an instance of the Account class*
Methods

Methods are functions defined in the suite of a class statement

```python
if amount > self.balance:
```

self should always be bound to an instance of the Account class
Methods

Methods are functions defined in the suite of a class statement

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

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

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

return 'Insufficient funds'
```

self should always be bound to an instance of the Account class
Methods

Methods are functions defined in the suite of a class statement

```python
self should always be bound to an instance of the Account class

self.balance = self.balance - amount
```
Methods

Methods are functions defined in the suite of a class statement

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

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

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

self should always be bound to an instance of the Account class
Methods

Methods are functions defined in the suite of a class statement

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

def deposit(self, amount):
    self.balance = self.balance + amount
    return self.balance

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

These def statements create function objects as always, but their names are bound as attributes of the class
Methods

Methods are functions defined in the suite of a class statement.

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

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

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

These `def` statements create function objects as always, but their names are bound as attributes of the class.

`self` should always be bound to an instance of the Account class.
Invoking Methods
Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state.
Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state.

class Account:
    ...
    def deposit(self, amount):
        self.balance = self.balance + amount
    return self.balance
Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state.

```python
class Account:
    ...
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Defined with two parameters
Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state.

```python
class Account:
    ...
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Dot notation automatically supplies the first argument to a method.
Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state.

class Account:
    ...
def deposit(self, amount):
      self.balance = self.balance + amount
      return self.balance

Dot notation automatically supplies the first argument to a method.

>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state.

class Account:
    ...
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

Dot notation automatically supplies the first argument to a method.

```python
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
```
Invoking Methods

All invoked methods have access to the object via the self parameter, and so they can all access and manipulate the object's state.

```python
class Account:
    ...
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```

Dot notation automatically supplies the first argument to a method.

```python
>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
100
```
Dot Expressions
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>
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.

<expression> . <name>

The <expression> can be any valid Python expression.

The <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 looked up by <name> in the object that is the value of the <expression>.
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.

<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 looked up by \text{<name>} in the object that is the value of the \text{<expression>}. 

\text{tom_account.deposit(10)}

\text{Dot expression} \quad \text{Call expression}
Dot Expressions

Objects receive messages via dot notation.

Dot notation accesses attributes of the instance or its class.

\(<expression> \cdot <name>\)

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)
Attributes

(Demo)
Accessing Attributes
Accessing Attributes

Using `getattr`, we can look up an attribute using a string.
Accessing Attributes

Using `getattr`, we can look up an attribute using a string

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

Using `getattr`, we can look up an attribute using a string.

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

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

Using `getattr`, we can look up an attribute using a string

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

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

`getattr` and `hasattr` look up a name in the same way.
Accessing Attributes

Using `getattr`, we can look up an attribute using a string

```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

```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

```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
Methods and Functions

Python distinguishes between:
Methods and Functions

Python distinguishes between:

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

Python distinguishes between:

• *Functions*, which we have been creating since the beginning of the course, and
• *Bound methods*, which couple together a function and the object on which that method will be invoked.
Methods and Functions

Python distinguishes between:

- **Functions**, which we have been creating since the beginning of the course, and
- **Bound methods**, which couple together a function and the object on which that method will be invoked.

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

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

```python
>>> type(Account.deposit)
```
Methods and Functions

Python distinguishes between:

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

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

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

```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'>
```
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
>>> 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(1003)
2014
```
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(1003)
2014
```

*Function*: all arguments within parentheses
### Methods and Functions

Python distinguishes between:

- **Functions**, which we have been creating since the beginning of the course, and
- **Bound methods**, which couple together a function and the object on which that method will be invoked.

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

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

```python
>>> Account.deposit(tom_account, 1001)
1011
>>> tom_account.deposit(1003)
2014
```

**Function**: all arguments within parentheses

**Method**: One object before the dot and other arguments within parentheses
Looking Up Attributes by Name

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

<expression> . <name>

To evaluate a dot expression:
Looking Up Attributes by Name

To evaluate a dot expression:

1. Evaluate the `<expression>` to the left of the dot, which yields the object of the dot expression.
Looking Up Attributes by Name

To evaluate a dot expression:

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

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

<expression> . <name>

To evaluate a dot expression:

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

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

3. If not, <name> is looked up in the class, which yields a class attribute value.
Looking Up Attributes by Name

To evaluate a dot expression:

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

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

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

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

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

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

class Account:

    interest = 0.02  # A class attribute

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

    # Additional methods would be defined here
Class Attributes

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

class Account:

    interest = 0.02  # A class attribute

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

        # Additional methods would be defined here

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

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

class Account:

    interest = 0.02  # A class attribute

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

    # Additional methods would be defined here

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

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

class Account:
    interest = 0.02  # A class attribute

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

    # Additional methods would be defined here

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

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

class Account:

    interest = 0.02  # A class attribute

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

    # Additional methods would be defined here

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

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

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

class Account:

    interest = 0.02  # A class attribute

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

    # Additional methods would be defined here

>>> tom_account = Account('Tom')
>>> jim_account = Account('Jim')
>>> tom_account.interest
0.02  # The interest attribute is not part of the instance; it's part of the class!
>>> jim_account.interest
0.02
Attribute Assignment
Assignment to Attributes
Assignment to Attributes

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

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

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

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

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

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

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

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

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

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

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

```python
class Account:
    interest = 0.02
    def __init__(self, holder):
        self.holder = holder
        self.balance = 0
...
tom_account = Account('Tom')
tom_account.interest = 0.08
```
Assignment to Attributes

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

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

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

tom_account = Account('Tom')

tom_account.interest = 0.08
Assignment to Attributes

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

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

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

tom_account.interest = 0.08

This expression evaluates to an object

But the name ("interest") is not looked up
Assignment to Attributes

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

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

```python
class Account:
    interest = 0.02
    def __init__(self, holder):
        self.balance = 0
        self.holder = holder
...
tom_account = Account('Tom')
tom_account.interest = 0.08
```

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

This expression evaluates to an object

But the name ("interest") is not looked up
Assignment to Attributes

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

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

```python
class Account:
    interest = 0.02
    def __init__(self, holder):
        self.holder = holder
        self.balance = 0
...
tom_account = Account('Tom')
```
Assignment to Attributes

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

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

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

tom_account = Account('Tom')
```

**Instance Attribute Assignment**

```
tom_account.interest = 0.08
```

**Class Attribute Assignment**

```
Account.interest = 0.04
```

This expression evaluates to an object

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

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

Account class attributes

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

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

Account class attributes:
- interest: 0.02
- (withdraw, deposit, __init__)
Attribute Assignment Statements

Account class attributes

interest: 0.02
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'

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

Account class attributes

interest: 0.02
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'

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

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

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

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

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

Account class attributes

interest: 0.02
(withdraw, deposit, __init__)

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

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

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

Account class attributes

interest: 0.02
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'

Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

Account class attributes

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

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

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

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

Account class attributes

interest: 0.02 0.04
(withdraw, deposit, __init__)

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

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

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

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

Account class attributes

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

Instance attributes of `jim_account`

- balance: 0
- holder: 'Jim'

Instance attributes of `tom_account`

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

Account class attributes

interest: 0.02  0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder:  'Jim'
interest: 0.08

Instance attributes of tom_account

balance: 0
holder:  'Tom'

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

>>> jim_account.interest = 0.08
Attribute Assignment Statements

Account class attributes

interest: 0.02  0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

Account class attributes

interest: 0.02 0.04
(withdraw, deposit, __init__)

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

Instance attributes of tom_account
balance: 0
holder: 'Tom'
interest: 0.04

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

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

Account class attributes

interest: 0.02 0.04
(withdraw, deposit, __init__)

Instance attributes of jim_account

balance: 0
holder: 'Jim'
interest: 0.08

Instance attributes of tom_account

balance: 0
holder: 'Tom'

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

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

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

```
Instance attributes of tom_account:
balance: 0
holder: 'Tom'
interest: 0.04
```

Account class attributes:
interest: 0.02 0.04 0.05
(withdraw, deposit, __init__)

21
**Attribute Assignment Statements**

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

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

Instance attributes of `tom_account`
- balance: 0
- holder: 'Tom'
- interest: 0.02

Account class attributes:
- interest: 0.02, 0.04, 0.05 (withdraw, deposit, __init__)
Attribute Assignment Statements

- `jim_account = Account('Jim')`
- `tom_account = Account('Tom')`
- `Account.interest = 0.04`
- `tom_account.interest = 0.04`
- `jim_account.interest = 0.04`
- `jim_account.interest = 0.08`
- `tom_account.interest = 0.05`
- `jim_account.interest = 0.08`

**Account class attributes**
- `interest: 0.02 0.04 0.05 (withdraw, deposit, __init__)`

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

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