Announcements
Announcements

• Homework 4 due Tuesday 10/8 @ 11:59pm.
Announcements

• Homework 4 due Tuesday 10/8 @ 11:59pm.

• Project 2 due Thursday 10/10 @ 11:59pm.
Announcements

• Homework 4 due Tuesday 10/8 @ 11:59pm.
• Project 2 due Thursday 10/10 @ 11:59pm.
• Homework 5 due Tuesday 10/15 @ 11:59pm.
Announcements

• Homework 4 due Tuesday 10/8 @ 11:59pm.
• Project 2 due Thursday 10/10 @ 11:59pm.
• Homework 5 due Tuesday 10/15 @ 11:59pm.
• Extra reader office hours this week in 405 Soda:
Announcements

- Homework 4 due Tuesday 10/8 @ 11:59pm.
- Project 2 due Thursday 10/10 @ 11:59pm.
- Homework 5 due Tuesday 10/15 @ 11:59pm.

Extra reader office hours this week in 405 Soda:

  - Tuesday 6–8pm, Wednesday 5:30–7pm, Thursday 5–7pm
Announcements

• Homework 4 due Tuesday 10/8 @ 11:59pm.
• Project 2 due Thursday 10/10 @ 11:59pm.
• Homework 5 due Tuesday 10/15 @ 11:59pm.
• Extra reader office hours this week in 405 Soda:
  ▪ Tuesday 6–8pm, Wednesday 5:30–7pm, Thursday 5–7pm
  ▪ (You can also go to regular office hours with questions about your project.)
Announcements

• Homework 4 due Tuesday 10/8 @ 11:59pm.

• Project 2 due Thursday 10/10 @ 11:59pm.

• Homework 5 due Tuesday 10/15 @ 11:59pm.

• Extra reader office hours this week in 405 Soda:
  ■ Tuesday 6–8pm, Wednesday 5:30–7pm, Thursday 5–7pm
  ■ (You can also go to regular office hours with questions about your project.)

• Guest lecture on Wednesday 10/9, Peter Norvig on Natural Language Processing in Python.
Announcements

• Homework 4 due Tuesday 10/8 @ 11:59pm.
• Project 2 due Thursday 10/10 @ 11:59pm.
• Homework 5 due Tuesday 10/15 @ 11:59pm.
• Extra reader office hours this week in 405 Soda:
  ▪ Tuesday 6–8pm, Wednesday 5:30–7pm, Thursday 5–7pm
  ▪ (You can also go to regular office hours with questions about your project.)
• Guest lecture on Wednesday 10/9, Peter Norvig on Natural Language Processing in Python.
  ▪ No video (except a screencast). Come to Wheeler!
Object-Oriented Programming
Object-Oriented Programming
Object-Oriented Programming

A method for organizing modular programs
Object-Oriented Programming

A method for organizing modular programs

• Abstraction barriers
Object-Oriented Programming

A method for organizing modular programs

- Abstraction barriers
- Bundling together information and related behavior
Object-Oriented Programming

A method for organizing modular programs

- Abstraction barriers
- Bundling together information and related behavior

A metaphor for computation using distributed state
Object-Oriented Programming

A method for organizing modular programs

• Abstraction barriers
• 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 modular programs
- Abstraction barriers
- 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 modular programs

- Abstraction barriers
- 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 modular programs

• Abstraction barriers
• 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 modular programs
• Abstraction barriers
• 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 modular programs
• Abstraction barriers
• 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 modular programs

• Abstraction barriers
• 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 modular programs
• Abstraction barriers
• 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 modular programs
- Abstraction barriers
- 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 modular programs

• Abstraction barriers
• 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 modular programs

• Abstraction barriers
• 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 modular programs
- Abstraction barriers
- 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 modular programs
• Abstraction barriers
• 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
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.
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')
```
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'
```
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 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
```

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

```python
>>> a.deposit(15)
15
```
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
```

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

```python
>>> a.deposit(15)
15
>>> a.withdraw(10)
5
```
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.

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

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

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

**Better idea:** All bank accounts share a "withdraw" method and a "deposit" method.

```python
>>> a.withdraw(10)
'Insufficient funds'
```
Class Statements
The Class Statement
The Class Statement

class <name>:
    <suite>
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.
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.

Statements in the `<suite>` create attributes of the class.
The Class Statement

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

Statements in the `<suite>` create attributes of the class.
The Class Statement

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

The suite is executed when a class statement is evaluated.

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

Statements in the `<suite>` create attributes of the class.

As soon as an instance is created, it is passed to `__init__`, which is an attribute of the class called the **constructor method**.
The Class Statement

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

The suite is executed when a class statement is evaluated.

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

Statements in the `<suite>` create attributes of the class.

As soon as an instance is created, it is passed to `__init__`, which is an attribute of the class called the **constructor method**.

```
class Account:
```
The Class Statement

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

Statements in the `<suite>` create attributes of the class.

As soon as an instance is created, it is passed to `__init__`, which is an attribute of the class called the *constructor method*.

```python
class Account:
    def __init__(self, account_holder):
```
The Class Statement

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

Statements in the `<suite>` create attributes of the class.

As soon as an instance is created, it is passed to `__init__`, which is an attribute of the class called the **constructor method**.

```python
class Account:
    def __init__(self, account_holder):
        self.balance = 0
```
A class statement **creates** a new class and **binds** that class to `<name>` in the first frame of the current environment.

Statements in the `<suite>` create attributes of the class.

As soon as an instance is created, it is passed to `__init__`, which is an attribute of the class called the *constructor method*.

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

**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')
```
Initialization

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

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

When a class is called:
Initialization

**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:
Initialization

**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:
When a class is called:

1. A new instance of that class is created:

2. The constructor `__init__` 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.

**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')
```
Initialization

**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 constructor `__init__` 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
```
Initialization

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

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

When a class is called:

1. A new instance of that class is created:

2. The constructor `__init__` 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
```
Initialization

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 constructor `__init__` 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
```
Initialization

**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:

```
{balance: 0}
```

2. The constructor `__init__` 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
```
Initialization

**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
   {balance: 0, holder: 'Jim'}
   ```

2. The constructor `__init__` 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
   ```
Initialization

**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:

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

2. The constructor `__init__` 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.

{balance: 0, holder: 'Jim'}
Initialization

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'
>>> a.balance
0
```

When a class is called:

1. A new instance of that class is created:

2. The constructor `__init__` 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 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')
```

Identity testing is performed by "is" and "is not" operators:
Object Identity

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

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

Identity testing is performed by "is" and "is not" operators:

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

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:

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

Identity testing is performed by "is" and "is not" operators:

```
>>> 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')
```

Identity testing is performed by "is" and "is not" operators:

```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 defined in the suite of a class statement.
Methods

Methods are defined in the suite of a class statement

\begin{verbatim}
class Account:
\end{verbatim}
Methods

Methods are defined in the suite of a class statement

class Account:
    def __init__(self, account_holder):


Methods

Methods are defined in the suite of a class statement

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

Methods are defined in the suite of a class statement

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

Methods are 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):
Methods

Methods are 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
Methods

Methods are 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
Methods

Methods are 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
    def withdraw(self, amount):
Methods

Methods are 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:
```


Methods

Methods are 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

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

Methods are 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

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

Methods are 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

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

Methods are 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
    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.
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.

```python
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.

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.

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.

```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
```
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
```
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>
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.

<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 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 \(<expression>\).
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>.

(Demo)
Attributes
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
g>>> getattr(tom_account, 'balance')
10
```
**Accessing Attributes**

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

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

>>> 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 dot expressions 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

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

global

`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
```

global `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
```

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

global `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.

Object + Function = 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.

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

```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(1000)
2011
```
Looking Up Attributes by Name

<expression> . <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 \(<\text{expression}>\) to the left of the dot, which yields the object of the dot expression.
Looking Up Attributes by Name

<expression> . <name>

To evaluate a dot expression:

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

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

<expression> . <name>

To evaluate a dot expression:

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

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

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

<expression> . <name>

To evaluate a dot expression:

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

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

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

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

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

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

class Account:
    interest = 0.02  # A class attribute
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    # Additional methods would be defined here
Class Attributes

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

```python
class Account:
    interest = 0.02  # A class attribute
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
        # Additional methods would be defined here

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

Class Attributes

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

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
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
>>> 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:
    interest = 0.02  # A class attribute
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    # Additional methods would be defined here

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

*interest* is not part of the instance that was somehow copied from the class!
Attribute Assignment
Assignment Statements and Attributes
Assignment Statements and Attributes

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

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

- If the object is an instance, then assignment sets an instance attribute
- If the object is a class, then assignment sets a class attribute
Assignment Statements and 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
>>> jim_account = Account('Jim')
```
Assignment Statements and 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
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
```
Assignment Statements and 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
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
```
Assignment Statements and 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
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
```
Assignment Statements and 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
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
```
Assignment Statements and 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
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
```
Assignment Statements and 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
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest = 0.02
>>> jim_account.interest = 0.02
>>> Account.interest = 0.04
```
Assignment Statements and 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
>>> 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
```
Assignment Statements and 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
>>> 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
```
Assignment Statements and 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
>>> 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
```
Assignment Statements and 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
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest = 0.02
>>> jim_account.interest
0.02
>>> jim_account.interest = 0.08
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
```
Assignment Statements and 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
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest = 0.02
>>> jim_account.interest
0.02
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
```
Assignment Statements and 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
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
```
Assignment Statements and 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
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest = 0.02
>>> jim_account.interest
0.02
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> Account.interest = 0.04
>>> tom_account.interest
0.04
```
Assignment Statements and 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
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest = 0.02
>>> jim_account.interest = 0.02
>>> jim_account.interest = 0.08
>>> tom_account.interest = 0.04
>>> Account.interest = 0.04
>>> tom_account.interest = 0.04
```
Assignment Statements and 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
>>> 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
>>> Account.interest = 0.05
```
Assignment Statements and 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
>>> 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
>>> Account.interest = 0.05
>>> tom_account.interest = 0.04
```
Assignment Statements and 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
>>> 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
>>> Account.interest = 0.05
>>> jim_account.interest = 0.08
>>> jim_account.interest
0.08
>>> tom_account.interest
0.04
>>> Account.interest = 0.05
>>> tom_account.interest
0.05
```
Assignment Statements and 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
>>> 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 = 0.08
>>> Account.interest = 0.05
>>> tom_account.interest = 0.05
>>> jim_account.interest = 0.08
```
Assignment Statements and 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
globals()  # <clear>

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 = 0.08
```

```python
globals()  # <clear>

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