Announcements
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
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

Diagram:
- John's Account
  - Withdraw $10
  - Deposit $10
- Jack's Account
  - Withdraw $10
  - Deposit $10
- John
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
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('John')
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('John')
```
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('John')
```
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('John')
>>> a.holder
'John'
```
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('John')
>>> a.holder
'John'
>>> a.balance
0
```
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('John')
>>> a.holder
'John'
>>> 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('John')
>>> a.holder
'John'
>>> a.balance
0
>>> a.deposit(15)
15
```

**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('John')
>>> a.holder
'John'
>>> 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

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

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

```python
>>> 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
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('John')
>>> a.holder
'John'
>>> 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.

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

```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).

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

>>> Clown.nose
'big and red'
```
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'
>>> 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'>
```
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).

>>> 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.
**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

```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

```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
```

`__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
>>> 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
```

_balance: 0  _holder: 'Jim'

__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
>>> 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

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

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

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

```python
>>> a = Account('John')
>>> 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('John')
>>> 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('John')
>>> 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('John')
>>> 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
```
Object Identity

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

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

```python
class Account:
    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.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
```

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

Methods are functions defined in the suite of a class statement

```
if amount > 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 += amount
        return self.balance

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

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

return 'Insufficient funds'
Methods

Methods are functions defined in the suite of a class statement

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

`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
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.
```

return self.balance
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.

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.

```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.

```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)  # Invoked with one argument
100                         # Bound to self
```
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 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>

```python
tom_account.deposit(10)
```

Call expression
**Dot Expressions**

Objects receive messages via dot notation.

Dot notation accesses attributes of the instance or its class

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

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

(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

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

- 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
```

`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'>

>>> 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

Object + Function = Bound Method

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

```python
>>> Account.deposit(tom_account, 1001)
1011
>>> tom_account.deposit(1004)
2015
```
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(1004)
2015
```

*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(1004)
2015
```

*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

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

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.

```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
```
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

```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')
>>> 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
>>> jim_account.interest
0.02

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