

## Objects

---

# Announcements

# Object-Oriented Programming

## Object-Oriented Programming

---

A method for organizing programs

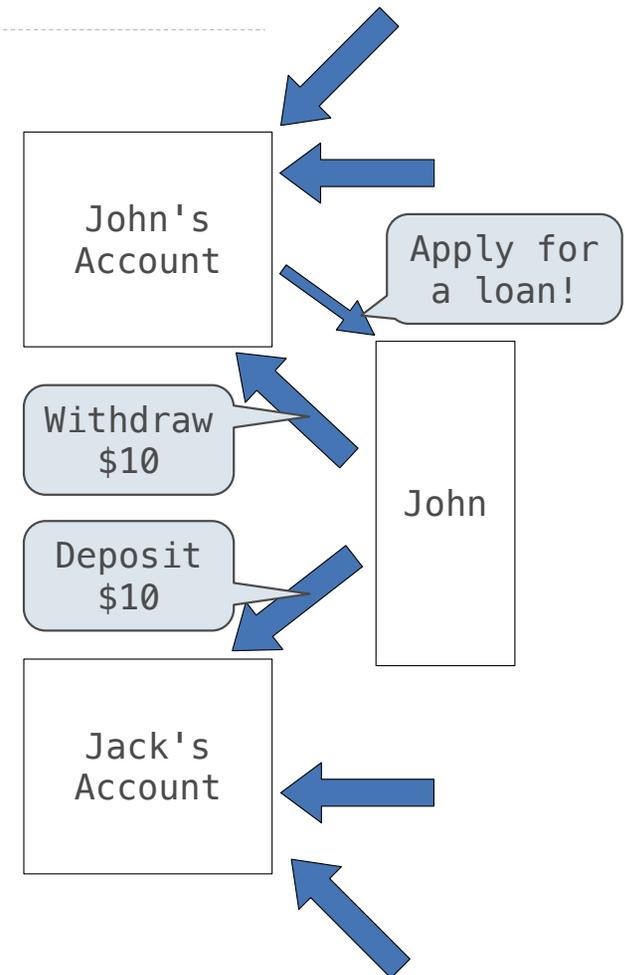
- Data abstraction
- Bundling together information and related behavior

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 describes the general behavior of 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')
>>> a.holder
'John'
>>> a.balance
0
```

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

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

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

```
>>> a.withdraw(10)
'Insufficient funds'
```

## Class Statements

## 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)

```
>>> 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

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

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

An account instance

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

`__init__` is called  
a constructor

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

## Object Identity

---

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

```
>>> 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.

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

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

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

```
>>> c = a
>>> c is a
True
```

## Methods

## Methods

---

Methods are functions defined in the suite of a class statement

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

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

    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

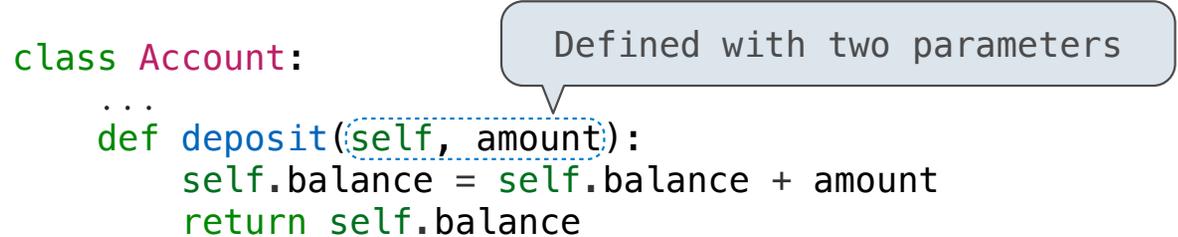
These def statements create function objects as always,  
but their names are bound as attributes of the class

## 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

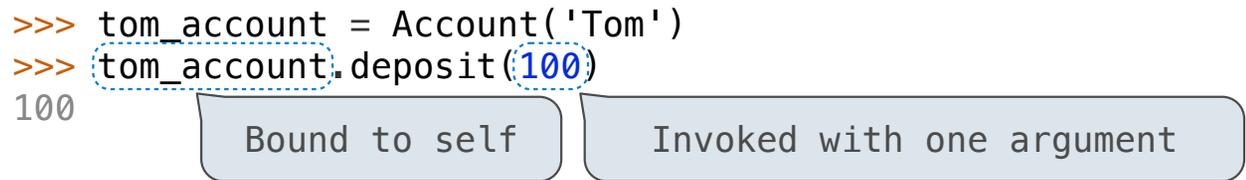
```
class Account:
    ...
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
```



The diagram shows the class definition for `Account`. A callout box labeled "Defined with two parameters" points to the `self` and `amount` parameters in the `deposit` method signature. The `self` parameter is also highlighted with a dashed blue box.

Dot notation automatically supplies the first argument to a method

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



The diagram shows the method invocation. A callout box labeled "Bound to self" points to `tom_account` in the `tom_account.deposit(100)` call. Another callout box labeled "Invoked with one argument" points to the `100` argument. The `tom_account` and `100` are also highlighted with dashed blue boxes.

## Dot Expressions

---

Objects receive messages via dot notation

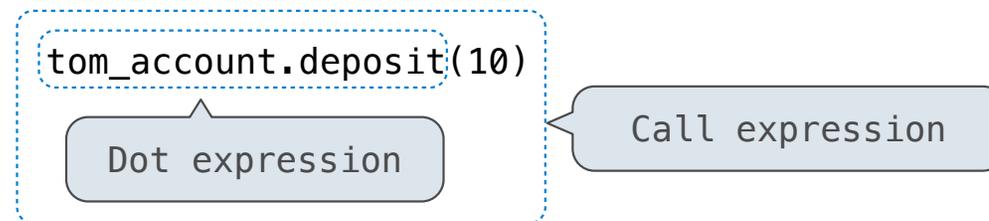
Dot notation accesses attributes of the instance or its class

`<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

(Demo)

## Accessing Attributes

---

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

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

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

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

Looking up an attribute name in an object may return:

- One of its instance attributes, or
- One of the attributes of its class

## Methods and Functions

---

Python distinguishes between:

- *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

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

```
>>> Account.deposit(tom_account, 1001)
1011
>>> tom_account.deposit(1007)
2018
```

**Function:** all arguments within parentheses

**Method:** One object before the dot and other arguments within parentheses

## 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 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!