Objects

Announcements

- Midterm grades released!
  - July 21st deadline to submit regrades
- Lab 07 and HW 04 released
- Cats due tomorrow, submit today for one bonus point
- Get excited for Ants
- OH is various locations, Woz and Warren, so check the calendar
- HW 3 Recovery released and will be due next Monday
- HW 2 Recovery due tonight

Object-Oriented Programming

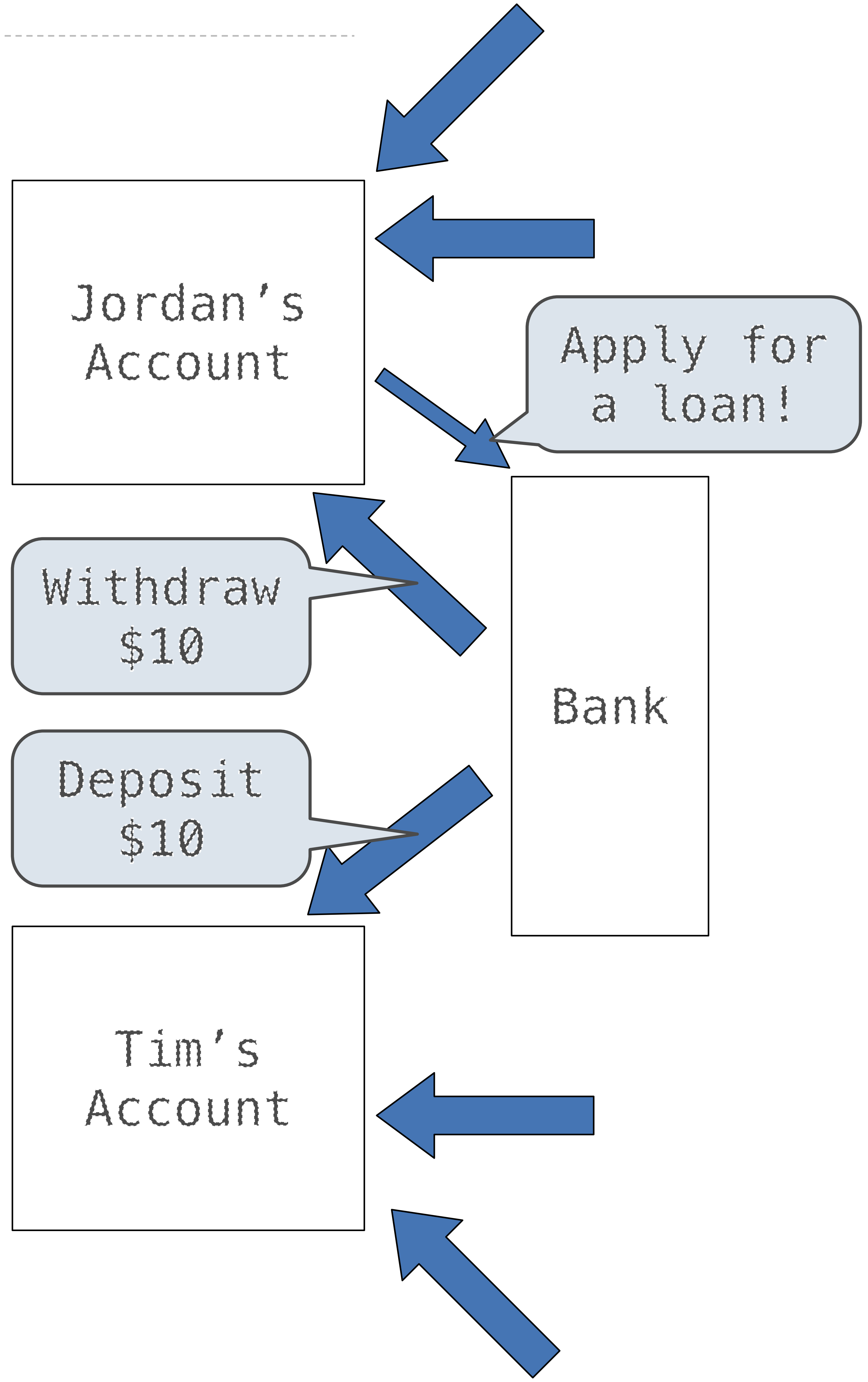
A method for organizing programs

- 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

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

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

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

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

Class vs. Object

- A class combines and abstracts data and functions
- An object is an instantiation of a class

Class Statements

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

```
>>> class House:
...     color = ‘red’
...     windows = 2
...
>>> House.color
‘red’
>>> House.windows
2
>>> House
<class ‘__main__.House'>
```

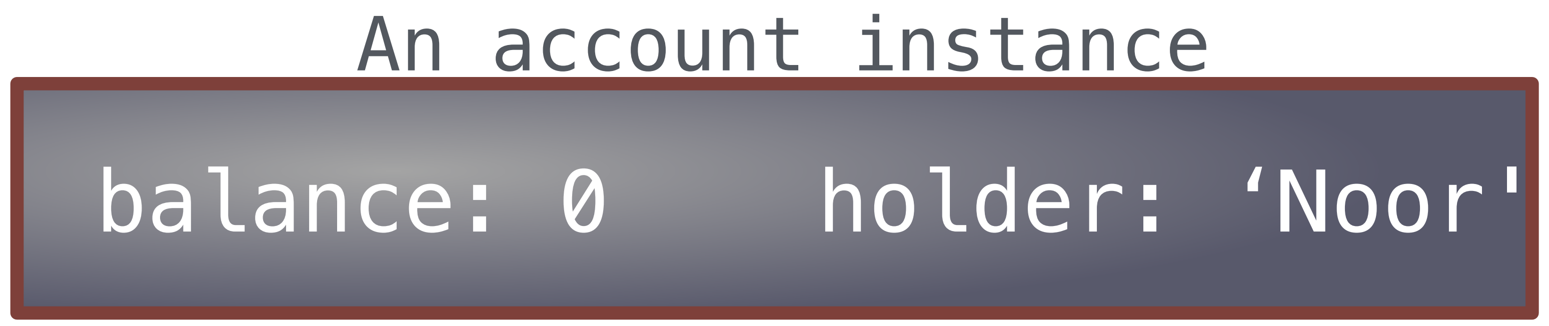
`

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('Noor')
>>> a.holder
'Noor'
>>> a.balance
0
```

When a class is called:

1. A new instance of that class is created:



An account instance

balance: 0 holder: 'Noor'

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

```
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('Tim')
>>> b = Account(‘Jordan')
>>> a.balance
0
>>> b.holder
‘Jordan'
```

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

Defined with two parameters

```
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
```

Bound to self

Invoked with one argument

Dot Expressions

Objects receive messages via dot notation

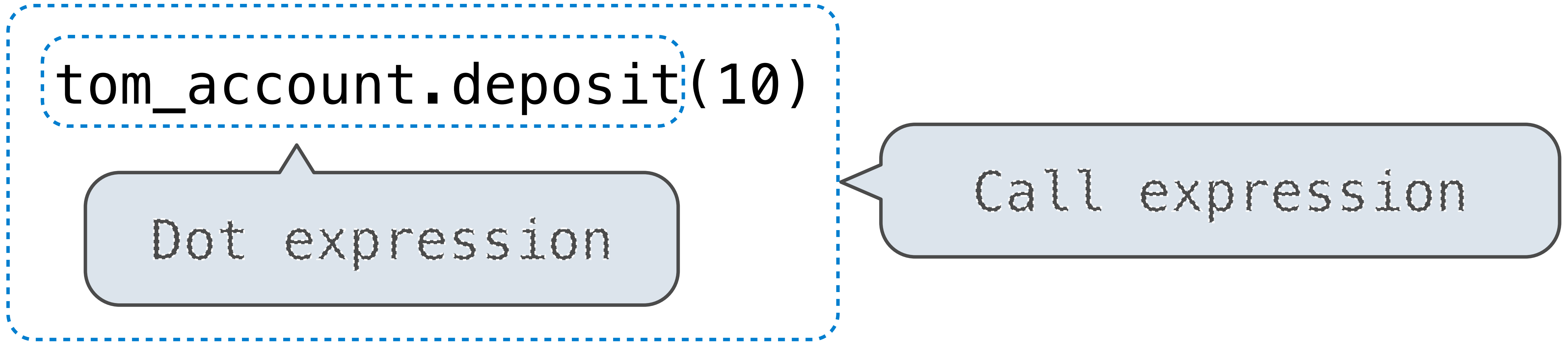
Dot notation accesses attributes of the instance or its class

```
<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)
```

Attributes

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

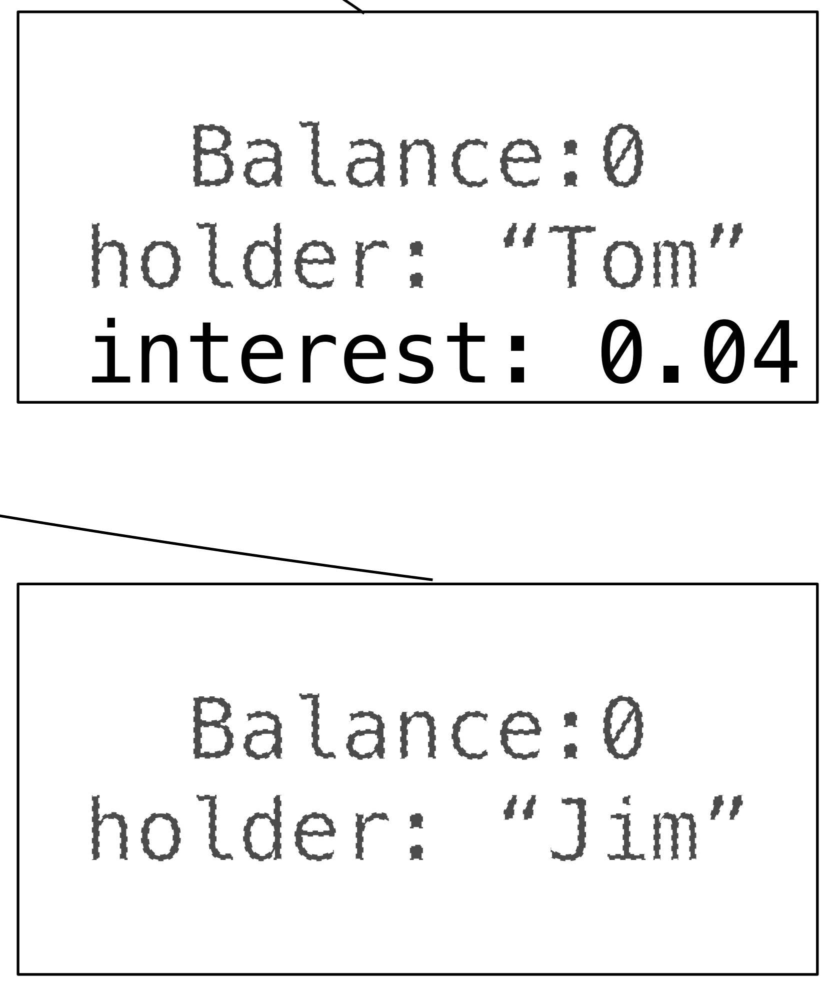
4. That value is returned unless it is a function, in which case a bound method is returned instead

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
               0.01
    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
>>> tom_account.interest = 0.04
>>> tom_account.interest
0.04
>>> Account.interest = 0.01
>>> tom_account.interest
0.04
>>> jim_account.interest
0.01
```



Balance:0
holder: “Tom”
interest: 0.04

Balance:0
holder: “Jim”

Break

Bouncing Balls

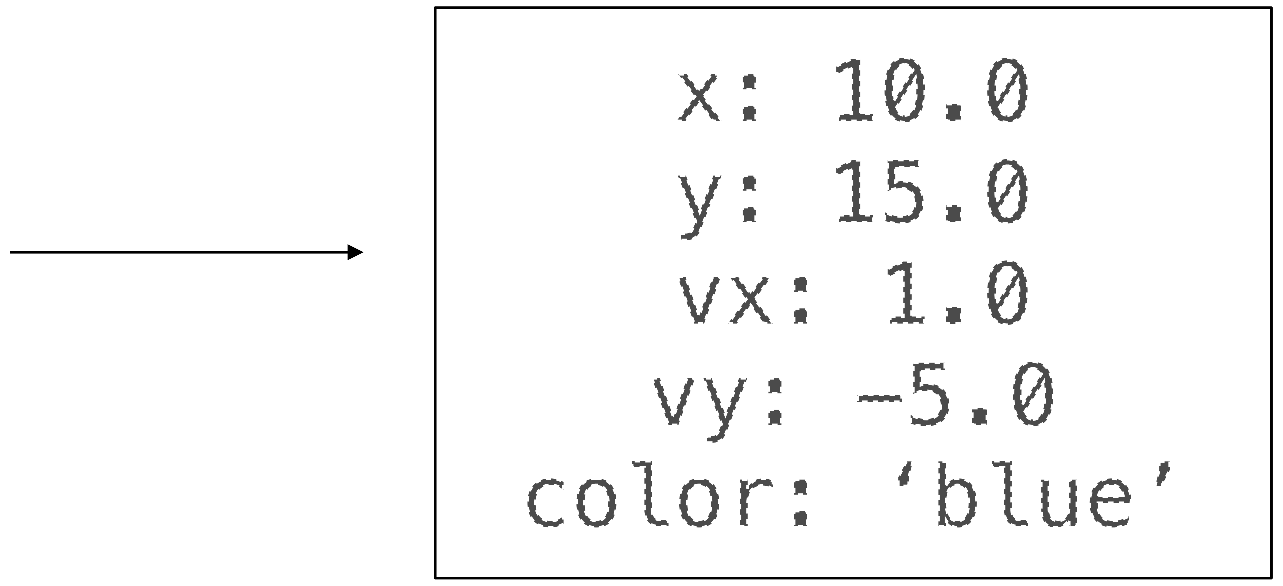
Ball Instance

```
class Ball:
    def __init__(self, start_x, start_y, start_v_x, start_v_y, color='blue'):
        # Ball location, velocity, and color
        self.x = start_x
        self.y = start_y
        self.v_x = start_v_x
        self.v_y = start_v_y
        self.color = color
```

```
...
```

1. Allocate memory for a Ball object
2. Initialize the Ball object with values
3. Return the Ball object

```
b1 = Ball(10.0, 15.0, 1.0, -5.0)
```



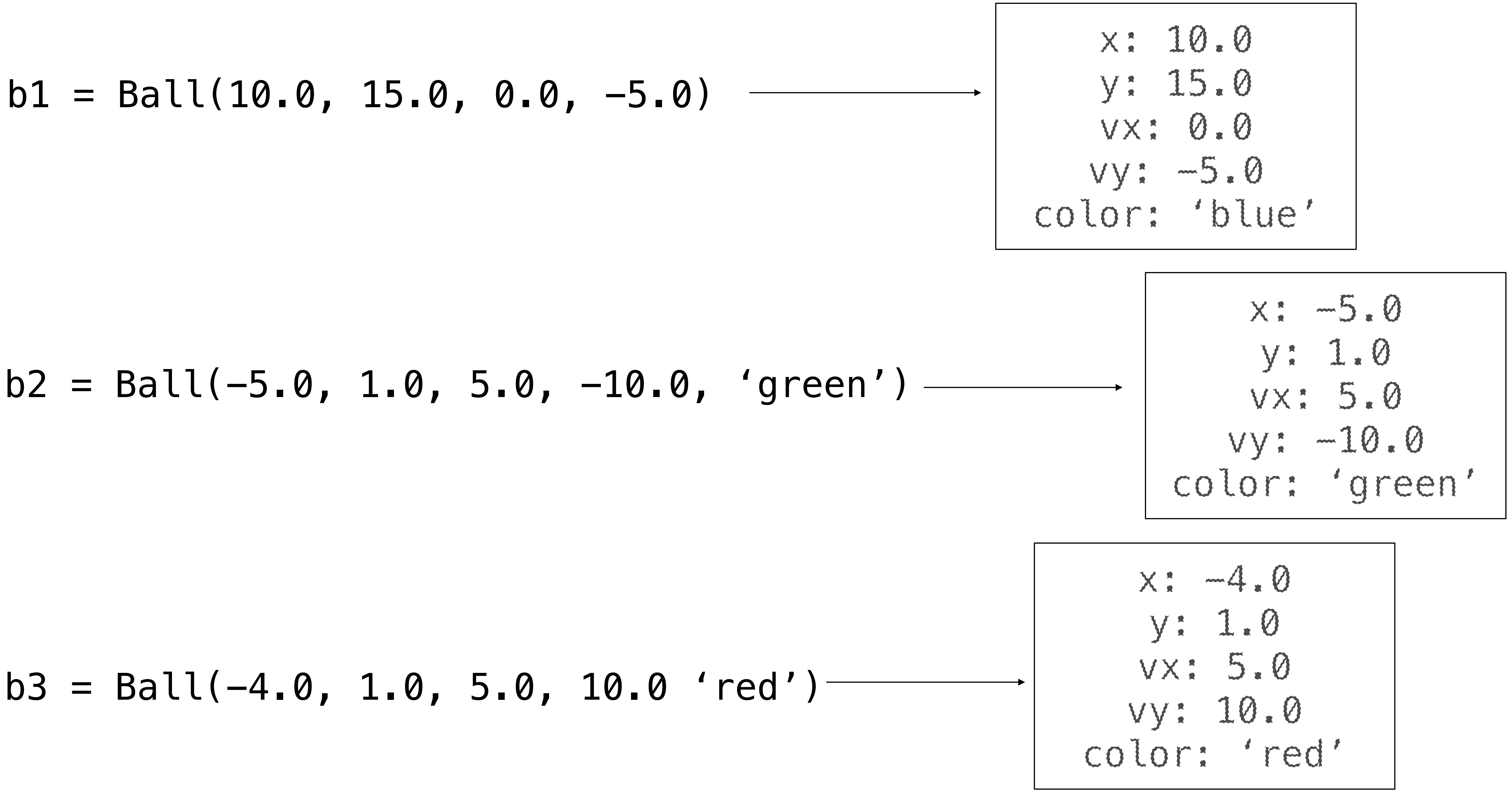
```
x: 10.0
y: 15.0
vx: 1.0
vy: -5.0
color: ‘blue’
```

```
>>> b1.x
10.0
>>> b1.update_position() # x+= vx
>>> b1.x
11.0
```

Ball Class

```python
class Ball:
    def __init__(self, start_x, start_y, start_v_x, start_v_y, color='blue'):
        # Ball location, velocity, and color
        self.x = start_x
        self.y = start_y
        self.v_x = start_v_x
        self.v_y = start_v_y
        self.color = color

    def update_position(self, timestep=1):
        self.x = self.x + timestep * self.v_x
        self.y = self.y - timestep * self.v_y
        if( self.y >= CANVAS_HEIGHT/2 - BALL_RADIUS): # bounce ball off floor
            self.v_y = -self.v_y
            self.y = self.y - timestep * self.v_y

    def update_velocity(self, timestep=1):
        self.v_y = self.v_y + timestep * EARTH_GRAVITY_ACCELERATION

    def animate_step(self, timestep=1):
        self.update_position(timestep)
        self.update_velocity(timestep)

    def draw_ball(self): # assumes canvas (D) has been created
        D.append(draw.Circle(self.x, self.y, BALL_RADIUS, fill=self.color))
```

Ball Class



```
b1 = Ball(10.0, 15.0, 0.0, -5.0)
```

```
x: 10.0
y: 15.0
vx: 0.0
vy: -5.0
color: ‘blue’
```

```
b2 = Ball(-5.0, 1.0, 5.0, -10.0, ‘green’)
```

```
x: -5.0
y: 1.0
vx: 5.0
vy: -10.0
color: ‘green’
```

```
b3 = Ball(-4.0, 1.0, 5.0, 10.0 ‘red’)
```

```
x: -4.0
y: 1.0
vx: 5.0
vy: 10.0
color: ‘red’
```

[bouncingballs.ipynb]

Multi-class programs

We can model objects interacting together!

Usually, we need more than one class of objects in our program to model its complexity!

[crowd.ipynb]